Partial Data Compression and Text Indexing via Optimal Suffix Multi-Selection

G. Franceschini[*] R. Grossi[†] S. Muthukrishnan[‡]

October 18, 2011

Abstract

Consider an input text string $T \equiv T[1,N]$ drawn from an unbounded alphabet, so text positions can be accessed using comparisons. We study *partial* computation in suffix-based problems for Data Compression and Text Indexing such as

- retrieve any segment of $K \leq N$ consecutive symbols from the Burrows-Wheeler transform of T, which is at the heart of the bzip2 family of text compressors, and
- retrieve any chunk of $K \leq N$ consecutive entries of the Suffix Array or the Suffix Tree, two popular Text Indexing data structures for T.

Prior literature would take $O(N \log N)$ comparisons (and time) to solve these problems by solving the *total* problem of building the entire Burrows-Wheeler transform or Text Index for T, and performing a post-processing to single out the wanted portion. The technical challenge is that the suffixes of interest are potentially of size $O(KN)$ and overlap in intricate ways: we have to use structural properties of these overlaps to avoid rescanning them repeatedly.

We introduce a novel adaptive approach to partial computational problems above, and solve both the partial problems in

$$O(K \log K + N)$$

comparisons and time, improving the best known running times of $O(N \log N)$ for $K = o(N)$.

These partial-computation problems are intimately related since they share a common bottleneck: the *suffix multi-selection* problem, which is to output the suffixes of rank $r_1, r_2, \ldots, r_K$ under the lexicographic order, where $r_1 < r_2 < \cdots < r_K$, $r_i \in [1,N]$. Special cases of this problem are well known: $K = N$ is the suffix sorting problem that is the workhorse in Stringology with hundreds of applications, and $K = 1$ is the recently studied suffix selection.

We show that suffix multi-selection can be solved in

$$\Theta\left(N \log N - \sum_{j=0}^{K} \Delta_j \log \Delta_j + N\right)$$

time and comparisons, where $r_0 = 0$, $r_{K+1} = N+1$, and $\Delta_j = r_{j+1} - r_j$ for $0 \leq j \leq K$. This is asymptotically optimal, and also matches the bound in [7] for multi-selection on atomic elements (not suffixes). Matching the bound known for atomic elements for strings is a long running theme and challenge from 70's, which we achieve for the suffix multi-selection problem. The partial suffix problems as well as the suffix multi-selection problem have many applications.

[*] Dipartimento di Informatica, Università "La Sapienza" di Roma, francesc@di.uniroma1.it

[†] Dipartimento di Informatica, Università di Pisa, grossi@di.unipi.it

[‡] Rutgers University, muthu@cs.rutgers.edu

1 Introduction

Consider an input text string $T \equiv T[1, N]$ and the set S of its suffixes $T_i \equiv T[i, N]$ $(1 \leq i \leq N)$ under the lexicographic order, where $T[N]$ is an endmarker symbol $\$$ smaller than any other symbol in T. The alphabet Σ from which the symbols in T are drawn is unbounded: as is standard in Stringology, we assume that any two symbols in Σ can only be compared and this takes $O(1)$ time. Hence, comparing symbolwise any two suffixes in S may require $O(N)$ time in the worst case.[1]

We study *partial computation* problems in Data Compression and Text Indexing for T where we want to quickly get a sense of the lexicographic distribution of the text suffixes.

Partial Data Compression. The Burrows-Wheeler transform L (a.k.a. BWT) [3] of text string T is at the heart of the bzip2 family of text compressors, and has many applications. The rth symbol in L is $T[j-1]$ if and only if T_j is the rth suffix in the sorting (except the borderline case $j = 1$, for which we take $T[N]$). There are now efficient methods that convert T to L and vice versa, taking $O(N \log N)$ time for unbounded alphabets in the worst case.

A partial compression problem is to consider a range $L' \equiv L[i..i+K-1]$ of K consecutive symbols in L. Can we compute L' efficiently? More precisely, can we compute L' without computing the entire L? This is an interesting building block for partial estimation of data compression ratio.

There is prior work that studies partial compression problems where a range $T[i, i+K-1]$ needs to be compressed (by Lempel-Ziv or Burrows-Wheeler or one of the other compression methods). This can be accomplished in $O(K \log K + N)$ time using off-the-shelf tools. Instead, what is interesting in our question above is that we seek a range in the *compressed* string L, and computing $L[i..i + K - 1]$ amounts to sorting a set of irregular, arbitrarily scattered suffixes $T_{j_1}, T_{j_2}, \ldots, T_{j_K}$ for which we do not know the positions $j_1, j_2, \ldots, j_K$ *a priori*!

Partial Text Indexing. Several text indexes, such as suffix arrays [21] and suffix trees [24, 29], are based on the lexicographic order of the suffixes in the text T and the longest common prefix (*lcp*) information among them. These can be computed in $O(N \log N)$ time using well-known algorithms that exploit properties of suffixes,[2] while the rest of the indexes can be easily built in $O(N)$ time.

We define a *K-partial text index* as a range $[i, i + K - 1]$ of the index (consecutive entries of the suffix array or leaves from the suffix tree): this corresponds to a sorted set of irregularly scattered suffixes $T_{j_1}, T_{j_2}, \ldots, T_{j_K}$ for which we do not know their positions $j_1, j_2, \ldots, j_K$ *a priori*, together with the length of their longest common prefix (*lcp*). The technical challenge here is similar to partial compression above, but additionally, we need to compute (*lcp*) information.

We refer the reader to Sections 2.1 and 2.2 for a more detailed discussion.

Basic questions. Can *partial suffix-based* computations like compression and indexing above, be solved more efficiently than solving the *total* problems? Besides the inherent interest in such problems and their structure that will let us parameterize their complexity in terms of $K \in [1, N]$, the main applied interest is that these partial problems give us a way to look at spots of a long string and get a sense for the complexity of data, be it for compressibility or performance of a full-text index.

The central technical challenge is the following. Given K symbols, sorting them is trivial. However, if we have to sort K arbitrary suffixes, they are of size $O(KN)$ in the worst case, and we can not afford to compare them symbolwise. In the worst case, it is better to perform a total sorting in $O(N \log N)$ time when $K = \Omega(\log N)$, as the suffixes overlap in arbitrary ways and we have to avoid rescanning the symbols repeatedly. Can we better this $O(N \log N)$ bound?

[1] Number of comparisons is asymptotically the same as the time for all the algorithms discussed throughout this paper, and hence we will use them interchangeably.

[2] It has an $O(N)$ time solution since 70's for constant-size alphabet Σ [24, 29] and, more recently, for (bounded universe) integer alphabet Σ [8]; otherwise, it uses $O(N \log |\Sigma|) = O(N \log N)$ comparisons in unbounded alphabets Σ.

Our results. We introduce a new adaptive approach to suffix sorting and order statistics.

Theorem 1 *Given a text T of length N, partial compression and partial text indexing problems can be solved in $O(K \log K + N)$ time.*

Hence for $K = o(N)$, partial compressing and text indexing problems can be solved asymptotically faster than their total counterparts. In particular, for $K = O(N/\log N)$, these partial problems can be solved in $O(N)$ time, which is quite rare in the comparison model.

We can also provide a bound for an arbitrary choice of K ranks $r_1, r_2, \ldots, r_K$ in the suffix order.

Theorem 2 *Given a text T of length N, partial compression and indexing can be solved using*

$$\Theta\left(N \log N - \sum_{j=0}^{K} \Delta_j \log \Delta_j + N\right) \tag{1}$$

time and comparisons, where $r_0 = 0$, $r_{K+1} = N + 1$, and $\Delta_j = r_{j+1} - r_j$ for $0 \leq j \leq K$; here, $1 \leq r_1 < r_2 < \cdots < r_K \leq N$ are the ranks of the suffixes involved in the output.

The algorithms behind Theorem 1 use an intermediate stage before applying the algorithms behind the more general Theorem 2, as otherwise the cost would be $O(K \log N + N)$ by choosing consecutive ranks $r_1, r_2, \ldots, r_K$ from the given range of values (i.e. $\Delta_j = 1$ for $1 \leq j < K$).

The above partial-computation problems share a common bottleneck: the *suffix multi-selection* problem, which is to output the suffixes of rank $r_1, r_2, \ldots, r_K$ under the lexicographic order, where $r_1 < r_2 < \cdots < r_K$, $r_i \in [1, N]$. Special cases of this problem are well known: $K = N$ is the standard suffix sorting problem, and $K = 1$ is the recently studied suffix selection for which $O(N)$-time comparison-based solutions are now known [11, 12]. We refer the reader to Section 2.3 for a more detailed discussion.

Theorem 3 *Given a text T of length N, the K text suffixes with ranks $r_1 < r_2 < \cdots < r_K$ (and the lcp's between consecutive suffixes) can be found within the bound stated in equation* (1)*. This is optimal.*

Related work. A long running theme in string matching has been matching for suffixes of a string, what one can do for atomic elements. The earliest suffix tree algorithms of 70's [24, 29] were interesting because they sort suffixes of a string over constant-sized alphabet in $O(N)$ time, matching the bucket sorting bound for N elements. However, it took lot longer to match the $O(N)$ time of radix sorting for strings over an integer alphabet in 90's [8], and in other computing models [9, 16]. For selection, the classic $O(N)$ time bound for atomic elements from 70's was matched only recently for string suffix selection [12, 11].

Similarly, it has been a technical challenge as we show here to match the multi-selection bound of atomic elements for string suffixes. Multi-selection for suffixes is not only interesting for reasons multi-selection problem in general is interesting, i.e., for statistical analysis of string suffixes, but also because it emerges naturally as the computational bottleneck of several problems like the partial compression and indexing problems described above, which have no natural counterpart in study of atomic elements.

For the sake of completeness, we recall that multi-selection of atomic elements includes basic problems such as sorting ($K = N$) and selection ($K = 1$) as special cases [18]. Selection algorithms that work in expected linear time [10, 13] or worst-case linear time [2] are now in textbooks. Multi-selection can also model intermediate problems between sorting and selection: for example, setting equally spaced r_i's, it corresponds to the quantile problem in Statistics. The asymptotically

original	*sorted*		*suffixes*	
		L	$i$	T_i
`ississippi$m`	`$mississipp`	`i`	**12**	`$`
`pi$mississip`	`i$mississip`	`p`	11	`i$`
`mississippi$`	`ippi$missis`	`s`	8	`ippi$`
`$mississippi`	`issippi$mis`	`s`	**5**	`issippi$`
`i$mississipp`	`ississippi$`	`m`	2	`ississippi$`
`ppi$mississi`	`mississippi`	`$`	1	`mississippi$`
`ippi$mississ`	`pi$mississi`	`p`	**10**	`pi$`
`sippi$missis`	`ppi$mississ`	`i`	9	`ppi$`
`ssippi$missi`	`sippi$missi`	`s`	7	`sippi$`
`issippi$miss`	`sissippi$mi`	`s`	**4**	`sissippi$`
`sissippi$mis`	`ssippi$miss`	`i`	6	`ssippi$`
`ssissippi$mi`	`ssissippi$m`	`i`	3	`ssissippi$`

Table 1: BWT L for the text $T =$ `mississippi$` and its relation with the sorted suffixes.

optimal number of comparison (and running time) is that in equation (1) as proved in [7].[3] Our suffix multi-selection algorithm is optimal, matching the lower bound even for atomic elements!

Paper organization. We first give more details on partial data compression, partial text indexing and suffix multiselection in Section 2, so as to relate the former two problems to the latter one. Then, the rest of the paper is devoted to suffix multi-selection (Theorem 3) with the following organization. We give the main ideas in and introduce the main concepts of subproblems and agglomerates, their data structures and algorithms, in Section 3. The top-level description of our multi-selection of suffixes is given in Section 4, and then all the implementation details are given in Section 5. Finally, we focus on the correctness and the analysis of the costs in Sections 6 and 7

2 Partial Data Compression, Partial Text Indexing, and Suffix Multi-Selection

We discuss here some of the new features that make the multi-selection problem on suffixes interesting and challenging, and focus on its applications to Data Compression and Text Indexing. To evaluate the benefits of our findings, we present some examples illustrating which tasks can be done optimally with known techniques and which cannot. At the same time we show that using our novel techniques, we can now perform optimally the latter tasks, which only had suboptimal algorithms so far.

2.1 Partial Data Compression

The Burrows-Wheeler transform (BWT) [3] is at the heart of the `bzip2` family of text compressors. Consider all the N circular shifts of the text $T =$ `mississippi$` as shown in the first column (*original*) of Table 1. Perform a lexicographic sorting of these shifts, as shown in the second column (*sorted*): if we single out the last symbol from each of the circular shifts in this order, we obtain a sequence L of N symbols that is called the BWT of T. Interestingly, not only we can recover T from L alone, but typically L is more compressible than T itself using 0th-order compressors (e.g. [22]). Its relation with suffix sorting is well known: the rth symbol in L is $T[j-1]$ if and only if T_j is

[3] The bound in (1) can be refined by studying the actual constant factors hidden in the Θ notation [15]. Several papers have studied other variations [14, 19, 20, 23, 26, 28].

the rth suffix in the sorting (except the borderline case $j = 1$, for which we take $T[N]$), as shown in the third column (*suffixes*).

Data compression ratio can be partially estimated by choosing a suitable sample L' of L for statistical purposes. There are several ways to make this choice, some are easy and some others are not, as we show next. It is easy to build L' if we take every other qth suffix in T. For example, $q = 3$ gives $L' = \texttt{\$pss}$ since we pick T_1, T_4, T_7, T_{10} and then perform their lexicographic sorting, namely, $T_1 < T_{10} < T_7 < T_4$. The latter is a simple variant of the standard suffix sorting and takes $O(N \log(N/q))$ time: the text T is conceptually partitioned as a sequence of N/q macro-symbols, where each macro-symbol is a segment of q actual symbols in T (could be less in the last one). Suffix sorting requires $O(N/q \log(N/q))$ macro-comparison, each involving q symbolwise comparisons, thus giving the above bound.

What if L' is chosen by taking every other qth symbol *directly* in L? Contrarily to the previous situation, here we guarantee a uniform sampling from L. For example, $q = 3$ gives $L' = \texttt{isps}$, which corresponds to selecting the suffixes $T_{12} < T_5 < T_{10} < T_4$ as shown in boldface in Table 1. Here comes a crucial observation: even though we sample from L with regularity, the starting positions of the chosen suffixes from the input text T form an *irregular* pattern and are difficult to predict without suffix sorting. We are not aware of any better approach other than performing a full execution of suffix sorting and, then, making a post-processing to single out every other qth sorted item. This yields a suboptimal cost of $O(N \log N)$ which should be compared to the $O(N \log(N/q)) = O(N \log K)$ cost in equation (1) by setting $\Delta_j = q$ for $j < K$ and $\Delta_K \leq q + 1$. In general, specifying ranks $r_1, r_2, \ldots, r_K$ gives the sample made up of the r_1th, r_2th, …, r_Kth symbols of L: using the algorithm giving the cost in (1), we can look at specific parts of the compressed string, without paying the full suffix sorting cost. We refer the reader to Theorem 2.

An intriguing situation arises when we consider just a segment of K consecutive symbols. Let us first consider a text segment $T[a, b]$, where $K = b - a + 1$ and $1 \leq a \leq b \leq N$. It takes $O(K \log K)$ time to perform a suffix sorting and compute the BWT of that segment alone; or $O(K \log K + N)$ time to sort the consecutive suffixes $T_a, T_{a+1}, \ldots, T_b$ whose starting positions lie in that segment, and then find their induced symbols inside L. Once again, these are simple variations of suffix sorting.

What if we want to compute only a segment $L[a, b]$ of $K = b - a + 1$ consecutive symbols instead of the whole L? For example, $L[3, 5] = \texttt{ssm}$ corresponds to suffixes $T_8 < T_5 < T_2$ in Table 1. In general, the starting positions of these suffixes form an irregular pattern and, as far as we know, the best that we can do is performing a *full* suffix sorting in $O(N \log N)$ time. Instead, setting $r_1 = a, r_2 = a + 1, \ldots, r_K = b$, we obtain that $\Delta_0 = a$, $\Delta_K = N + 1 - b$, and $\Delta_j = 1$ for $1 < j < K$. Hence, the cost implied by equation (1) is $O(K \log N + N)$, and the computed longest common prefixes in Theorem 3 will also provide the contexts for estimating the empirical entropy of L restricted to the segment $L[a, b]$. Even better, we can obtain $O(K \log K + N)$ using the same algorithm behind equation (1) and an analysis focussed on this special case (i.e. the wanted ranks form an interval of consecutive values, see Lemma 18). When $K = o(N)$, this compares favorably with the suboptimal $O(N \log N)$ cost of building L explicitly. We refer the reader to Theorem 1.

2.2 Partial Text Indexing

Several text indexes, such as suffix arrays [21] and suffix trees [24, 29], are based on the lexicographic order of the suffixes in the text T and the longest common prefix (*lcp*) information among them. Ultimately, the suffix sorting and the *lcp* information constitute the kernel upon which the rest of the index can be easily built in $O(N)$ time. An instance of suffix tree for the text $T = \texttt{mississippi\$}$ is shown in Figure 1(left), and consists of a compacted trie storing all the suffixes of T.

Based on the above observation, we define a *K-partial text index* as a subset of K suffixes plus their *lcp* information. Having this, we can build the suffix array or the suffix tree restricted to these

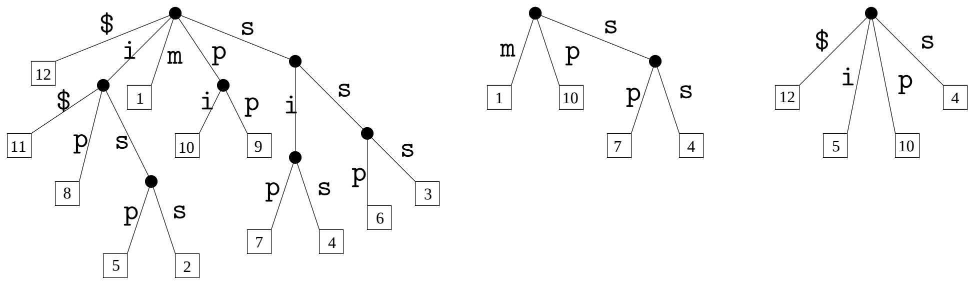


Figure 1: A suffix tree and two ways to sample it. Only the first symbol is shown on each labelled edge.

K suffixes in $O(K)$ time. Hence, the problem of building a K-partial text index tantamounts to performing a sorting of these K suffixes and finding their *lcp* information. We discuss two ways of choosing these K suffixes.

One possibility is sampling every other qth *suffix* in T, as shown in Figure 1(center) with $q = 3$. This is the *sampled* suffix tree introduced in [17], and its construction is a simple variant of the standard one and takes $O(N \log(N/q)) = O(N \log K)$ time: as previously mentioned, the text T is conceptually partitioned as a sequence of $K = N/q$ macro-symbols.

Another possibility is sampling every other qth *leaf* directly from the suffix tree, as shown in Figure 1(right) with $q = 3$. However, with the current techniques we have a suboptimal solution: build first the whole suffix tree in $O(N \log N)$ time and then perform a post-processing to select the wanted leaves and their ancestors (removing the possible unary nodes thus created). Using the algorithm behind equation (1) and the related longest common prefix information as a byproduct, we are able to build this K-partial index in $O(N \log(N/q)) = O(N \log K)$ time by fixing $\Delta_j = q$ for $j < K$ and $\Delta_K \leq q + 1$. Contrarily to the sampled suffix tree above, here the starting positions of the chosen suffixes form an *irregular* pattern even though we sample from the suffix tree with regularity.

In general, given a text T and its ranks $r_1, r_2, \ldots, r_K$, we want to build the K-partial text index for the suffixes having those ranks. For example, when employing a suffix tree, the K-partial text index gives the subtrie made up of the r_1th, r_2th, …, r_Kth leaves of the suffix tree for T. But we do not want to build that full suffix tree explictly in $O(N \log N)$ time. Using the algorithm behind equation (1), we can attain this goal. We refer the reader to Theorem 2.

A somewhat surprising situation arises when considering just a segment of K consecutive symbols. Consider first the K-partial index build on the text segment $T[a, b]$, where $K = b - a + 1$ and $1 \leq a \leq b \leq N$. It takes $O(K \log K)$ time to build the suffix array or the suffix tree for that segment alone, or $O(K \log K + N)$ time to build them for the consecutive suffixes $T_a, T_{a+1}, \ldots, T_b$ whose starting positions lie in that segment. Once again, these are simple variations of known algorithms.

This is not the case when computing just a segment of K consecutive entries in the suffix array, or just a chunk of K consecutive leaves from the suffix tree. Clearly, we do not want the suboptimal solution that builds the entire suffix array or suffix tree in $O(N \log N)$ time. Ours is a special case of K-partial text index, since the wanted suffixes have *consecutive ranks* $r_1 = a, r_2 = a+1, \ldots, r_K = b$. The $O(K \log N + N)$ cost implied by equation (1) can be refined (using Lemma 18) so that we obtain $O(K \log K + N)$, comparing favorably with the suboptimal cost of building the suffix array/tree entirely. In general, the task of computing only a chunk of K consective entries from the suffix array or the suffix tree falls within the following result. We refer the reader to Theorem 1.

2.3 Suffix Multi-Selection Problem

Given a set S of N elements from a total order $<$, the rank of $x \in S$ is $r \in [1, N]$ if x is the rth smallest in S, namely, $r = 1 + |\{y \in S | y < x\}|$. For integers $r_1 < r_2 < \cdots < r_K$, where each $r_i \in [1, N]$, the *multi-selection* problem is to select the elements of rank $r_1, r_2, \ldots, r_K$ from S.

Multi-selection includes basic problems such as sorting and selection as special cases. When $K = N$, it finds the ranks for all the elements, making it straightforward to arrange them in sorted order [18]. When $K = 1$, it corresponds to the standard selection problem: given an integer $r \in [1, N]$, the goal is to return the element of rank r in S. Selection algorithms that work in expected linear time [10, 13] or worst-case linear time [2] are now in textbooks. It can also model intermediate problems bewteen sorting and selection: for example, setting equally spaced r_i's, it corresponds to the quantile problem in Statistics. Multi-selection can thus arise in applications for partitioning the input, say for a recursive approach.

To the best of our knowledge the first algorithm for multi-selection was given in [7], thus establishing that the *asymptotically* optimal number of comparison (and running time) is that in equation (1), where $r_0 = 0$ and $r_{K+1} = N + 1$ and $\Delta_j \equiv r_{j+1} - r_j$ for $0 \leq j \leq K$. The formula in (1) can be intuitively read as follows. Find the Δ_0 smallest elements, then the Δ_1 next smallest elements, and so on, up to the last Δ_K ones. The resulting arrangement is almost sorted, and can be fully sorted by ordering each individual group of Δ_j elements independently in $\Theta(\Delta_j \log \Delta_j)$ time. Hence, take the total sorting cost of $\Theta(N \log N)$, subtract the cost of sorting each group, i.e. $\Theta(\sum_{j=0}^{K} \Delta_j \log \Delta_j)$, and add $\Theta(N)$ to read all the elements as a baseline. Note that rewriting (1) as $\Theta(N \sum_{j=0}^{K} (\Delta_j/N) \log(N/\Delta_j) + N)$, where $\sum_{j=0}^{K} \Delta_j = N + 1$, we can reformulate the bound in (1) as $\Theta(N(H_0 + 1))$ where $H_0 = -\sum_{j=0}^{K} p_j \log_2 p_j$ is the empirical 0th-order entropy where $p_j = \Delta_j/N$ is the empirical probability of having the jth group of size Δ_j.

The asymptotical optimality of (1) can be further refined by studying the actual constant factors hidden in the Θ notation. Any comparison-based algorithm must perform at least $B = N \log N - \sum_{j=0}^{K} \Delta_j \log \Delta_j - O(N)$ comparisons to solve multi-selection, and the algorithm in [15] is nearly optimal in this sense. It attains $B + O(N)$ expected comparisons and $B + o(B) + O(N)$ comparisons in the worst case, taking $O(B + N)$ running time. For the interested reader, other papers have studied further features in [14, 19, 20, 23, 26, 28].

We focus on the asymptotically optimality of the bound in (1) and call *optimal* an algorithm that asymptotically meets that bound, namely, $\Theta(B + N)$ in the worst case. Unless specified, the running time is always proportional to the number of pairwise comparisons between elements.

In this paper, we study the analog of the multi-selection problem in Stringology. Consider an input text string $T \equiv T[1, N]$ and the set S of its suffixes $T_i \equiv T[i, N]$ $(1 \leq i \leq N)$ under the lexicographic order. Let $T[N]$ be an endmarker symbol, denoted by $\$$, which is smaller than any other symbol in T. The alphabet Σ from which the symbols in T are drawn is unbounded and the comparison model is adopted: any two symbols in Σ can only be compared and this takes constant time. Hence, comparing symbolwise any two suffixes in S may requires $O(N)$ time in the worst case. Given ranks $r_1 < r_2 < \cdots < r_K$, the *suffix multi-selection* problem is to output the suffixes of rank $r_1, r_2, \ldots, r_K$ in S. Our main contribution is that we extend the asymptotically optimal bound in (1) to the suffixes in S. We refer the reader to Theorem 3.

Recall that the length of the *longest common prefix* of any two suffixes T_i and T_j is defined as the smallest $\ell \geq 0$ such that $T[i + \ell] \neq T[j + \ell]$. We also can obtain this info as a byproduct, which is useful for the problems mentioned in Sections 2.1 and 2.2.

3 Concepts, Definitions, and Main Ideas

For the given string $T \equiv T[1,N]$, let $T[N]$ be an endmarker symbol, denoted by \$, that is smaller than any symbol of the alphabet and does not appear elsewhere in T. Let $T_i \equiv T[i,N]$ be the i-th suffix of T, and $S = \{T_i \mid 1 \leq i \leq N\}$ be the set of all these suffixes. Given the set $\mathcal{R}$ of ranks $r_1 < r_2 < \cdots < r_K$, we want to select the suffixes $T_{i_1} < T_{i_2} < \cdots < T_{i_K}$ such that T_{i_j} has rank r_j in S, for $1 \leq j \leq K$. Since we already know that T_N is the smallest suffix of T, we can assume wlog that $r_1 > 1$. Also, we use $<$ and $\leq$ to denote string comparison according to the lexicographical order. For any two sets X, Y, notation $X < Y$ indicates that $x < y$ for any pair $x \in X, y \in Y$.

Our main goal is to prove Theorem 3, as this is the major obstacle when proving Theorems 1– 2. The tricks of the trade all rely on the following "golden rule" on the *lcp* information for the suffixes: For any integer $d > 0$, if $lcp(T_i, T_j) > \ell$ for some integer $\ell \geq d$, then $lcp(T_{i+d}, T_{j+d}) \geq \ell - d$. Thus, if T_i and T_j have been compared, the direct comparison of the first $\ell - d$ symbols of T_{i+d} and T_{j+d} can be avoided. Conversely, if T_{i+d} and T_{j+d} have been compared, the comparison of all but the first d symbols in T_i and T_j can be avoided.

Unfortunately, we cannot always rely on the golden rule here. When comparing T_i and T_j, we do not yet know whether or not both T_{i+d} and T_{j+d} will have to be compared (directly or indirectly by transitivity) for a choice of $d > 0$, or vice versa: simply put, we cannot predict at each stage of the computation whether the comparison between T_{i+d} and T_{j+d} will occur or not in the future.

3.1 Subproblems

At the beginning, all the suffixes form a single problem $S \equiv \{T_1, T_2, \ldots, T_n\}$. At the end, we want to obtain a partition of S into *subproblems*, namely, $S = S_1 \cup \{T_{i_1}\} \cup S_2 \cup \{T_{i_2}\} \cup \cdots \cup S_K \cup \{T_{i_K}\} \cup S_{K+1}$, such that S_j contains all the suffixes in S of rank r with $r_{j-1} < r < r_j$ for $1 \leq j \leq K+1$. Although each S_j is not internally sorted, still $S_1 < \{T_{i_1}\} < S_2 < \{T_{i_2}\} < \cdots < S_K < \{T_{i_K}\} < S_{K+1}$.

At an arbitrary stage of the computation, the suffixes in S and the ranks in $\mathcal{R}$ are partitioned amongst the *subproblems*. Let us call $\mathcal{P}_1 < \mathcal{P}_2 < \cdots < \mathcal{P}_z$ the subproblems in the current stage, where $S = \mathcal{P}_1 \cup \mathcal{P}_2 \cup \cdots \cup \mathcal{P}_z$, and let $\mathcal{R}(\mathcal{P}_i)$ be the set of ranks associated with subproblem $\mathcal{P}_i$, for $1 \leq i \leq z$. Namely, $r \in \mathcal{R}(\mathcal{P}_i)$ iff $\ell_i < r \leq \ell_i + |\mathcal{P}_i|$, where $\ell_i = \sum_{j<i} |\mathcal{P}_j|$ is the number of lexicographically smaller suffixes.

Status. A subproblem $\mathcal{P}_i$ is *solved* if $|\mathcal{P}_i| = |\mathcal{R}(\mathcal{P}_i)| = 1$, and *unsolved* if $|\mathcal{P}_i| > 1$ and $|\mathcal{R}(\mathcal{P}_i)| \geq 1$. A subproblem $\mathcal{P}_i$ and its suffixes are *exhausted* if $|\mathcal{R}(\mathcal{P}_i)| = 0$. A subproblem $\mathcal{P}_i$ and its suffixes are *degenerate* if each of these suffixes share an *empty* prefix with any of the wanted suffixes $T_{i_1} < T_{i_2} < \cdots < T_{i_K}$. (Note that (*i*) a degenerate subproblem is also exhausted and (*ii*) T_N is always degenerate since we assume $r_1 > 1$.)

A subproblem $\mathcal{P}_i$ is never merged with others and can only be refined into smaller subproblems. A partition of $\mathcal{P}_i$ into $\mathcal{P}_{i_1}, \ldots, \mathcal{P}_{i_p}$ is called a *refinement* if for any two $\mathcal{P}_{i_j}, \mathcal{P}_{i_{j'}}$ either $\mathcal{P}_{i_j} < \mathcal{P}_{i_{j'}}$ or $\mathcal{P}_{i_{j'}} < \mathcal{P}_{i_j}$: note that the refinement is a stronger notion than the partition, since it also takes into account of the lexicographic order among the suffixes.

During the computation a subproblem can be either *active* or *inactive.* If $|\mathcal{R}(\mathcal{P}_i)| \geq 1$ then $\mathcal{P}_i$ is active. An *active* subproblem is subjected to the *refinement*, whereas this does not hold anymore once it becomes *inactive.* If $\mathcal{P}_i$ is inactive then $|\mathcal{R}(\mathcal{P}_i)| = 0$ and thus it is exhausted. (The latter is a necessary condition: not all the exhausted subproblems are inactive.) Once a subproblem becomes inactive it will stay inactive until the end. Degenerate subproblems are inactive since the start. We give a complete characterization in Section 3.2.

Integer labels and neighborhood. Ideally, we would like to maintain the integer label $\ell_i = \sum_{j<i} |\mathcal{P}_j|$ for each subproblem $\mathcal{P}_i$ during the refining process. Using the integer labels, we can define a total order relation $\lhd$ and an equivalence relation $\simeq$ on the suffixes of T. Consider any two

suffixes $T_{i'}, T_{j'}$ and their subproblems' labels ℓ_i, ℓ_j. We have that $T_{i'} \lhd T_{j'}$ iff *(1)* $T[i'] \leq T[j']$ when both $T_{i'}$ and $T_{j'}$ are degenerate or *(2)* $\ell_i \leq \ell_j$ otherwise. Similarly, $T_{i'} \simeq T_{j'}$ iff *(1)* $T[i'] = T[j']$ when both T_i and T_j are degenerate or *(2)* $\ell_i = \ell_j$ otherwise. The total order $\lhd$ is consistent with the lexicographical order: if $T_{i'} \lhd T_{j'}$ and $T_{i'} \not\simeq T_{j'}$ then $T_{i'} < T_{j'}$. If the labels for two suffixes are known then comparing them according to $\lhd$ and $\simeq$ takes $O(1)$ time.

After an active subproblem $\mathcal{P}_i$ becomes inactive, it is possible that some of its suffixes are moved to form other inactive subproblems $\mathcal{P}_{i_j}$'s but we still need the value of ℓ_i. For this reason, we need to introduce a more general notion, that of *neighborhood* $\mathcal{N}_i = \mathcal{P}_i \cup \bigcup_{i=1}^{l} \mathcal{P}_{i_j}$, to preserve what was once an individual active subproblem $\mathcal{P}_i$. Summing up: if $\mathcal{P}_i$ is *active* then $\mathcal{N}_i = \mathcal{P}_i$; else, $\mathcal{N}_i \supseteq \mathcal{P}_i$. In any case, the reference label is $\ell_i = \sum_{\mathcal{N}_j < \mathcal{N}_i} |\mathcal{N}_j| = |\{T_j | T_j < T_i \text{ for any } T_i \in \mathcal{N}_i\}|$.

For the sake of description, each subproblem $\mathcal{P}_i$ or neighborhood $\mathcal{N}_i$, and by extension each suffix in them, is conceptually associated with an *α-string* α_i. If $\mathcal{P}_i$ is *non-degenerate* then T_j has α_i as prefix iff $T_j \in \mathcal{N}_i$. If $\mathcal{P}_i$ is *degenerate*, a weaker property holds since $|\alpha_i| = 0$: for any $\mathcal{P}_j$ not in $\mathcal{N}_i$, either $\mathcal{P}_j < \mathcal{N}_i$ or $\mathcal{N}_i < \mathcal{P}_j$. Observing that only the integer labels are used to compare suffixes according to $\lhd$ and $\simeq$, the α-strings will not be maintained during the computation. For the presentation in the paper, we will focus on subproblems rather than neighbors, keeping in mind that the label of an inactive subproblem is that defined for its neighborhood.

Rationale. We refine the subproblems as follows. We pick an unsolved subproblem $\mathcal{P}_i$ and refine it into smaller ones: We find the closest ranks r_j, r_{j+1} for $\mathcal{P}_i$ partitioning it "evenly," namely, $r_j \leq \ell_i + |\mathcal{P}_i|/2 \leq r_{j+1}$. Then, we select the suffixes $T_{i_j}, T_{i_{j+1}}$ with ranks r_j, r_{j+1} and partition $\mathcal{P}_i$ into three new subproblems according to T_{i_j} and $T_{i_{j+1}}$. The new subproblem with the middle suffixes is exhausted since it has no ranks associated (and will form S_{j+1}), while the other two subproblems are still unsolved. The goal is to reach a situation in which each subproblem $\mathcal{P}_i$ is either exhausted ($\mathcal{P}_i \equiv S_j$ for some j) or solved ($\mathcal{P}_i \equiv \{T_{i_j}\}$ for rank r_j and some j): namely, $S = \mathcal{P}_1 \cup \mathcal{P}_2 \cup \cdots \cup \mathcal{P}_z \equiv S_1 \cup \{T_{i_1}\} \cup S_2 \cup \{T_{i_2}\} \cup \cdots \cup S_K \cup \{T_{i_K}\} \cup S_{K+1}$ is the resulting refinement of the initial problem S. This scheme works if we suppose that S contains independent strings. Unfortunately, S contains the suffixes of T and so the rescanning cost is the main obstacle.

3.2 Agglomerates of subproblems

We group subproblems into agglomerates to model the interplay among suffixes that share the *same* α-string. We represent each agglomerate as a threaded dynamic tree where each node represents a subproblem (a subset of suffixes), as illustrated in the example of Fig. 2.

Dependency. For any two subproblems $\mathcal{P}_i, \mathcal{P}_j$, we say that $\mathcal{P}_i$ depends directly on $\mathcal{P}_j$ if the following hold: *(i)* $\mathcal{P}_i = \mathcal{P}_j$ or *(ii)* $\mathcal{P}_i \neq \mathcal{P}_j$ and for each $T_x \in \mathcal{P}_i$ we have that $T_{x+1} \in \mathcal{P}_j$. We extend this relation by transitivity, denoted $\sqsubseteq^+$, which is a partial order. When $\mathcal{P}_i \sqsubseteq^+ \mathcal{P}_j$ we say that $\mathcal{P}_i$ *depends on* $\mathcal{P}_j$. We extend this terminology to single suffixes: a suffix $T_i \in \mathcal{P}_{i'}$ *depends on* $T_j \in \mathcal{P}_{j'}$ (denoted by $T_i \sqsubseteq^+ T_j$) if $i \leq j$, $\mathcal{P}_{i'} \sqsubseteq^+ \mathcal{P}_{j'}$ and $T_x \notin \mathcal{P}_{i'} \cup \mathcal{P}_{j'}$, for $i < x < j$.

Partial order and tree representation. A set of subproblems is an *agglomerate* A if there exists $\mathcal{P}_i \in A$ such that $\mathcal{P}_j \sqsubseteq^+ \mathcal{P}_i$, for each $\mathcal{P}_j \in A$ (i.e. A has a maximum according to the partial order $\sqsubseteq^+$). The Hasse diagram according to $\sqsubseteq^+$ for an agglomerate A is a tree whose root is the *maximum* subproblem and the leaves are the *minimal* subproblems of A. Moreover, the children of an internal node are subproblems directly depending on it (see Fig. 2). We denote by $|A|$ the number of subproblems in A, and apply tree terminology to agglomerates. A suffix $T_x \in \mathcal{P}_i \in A$ $(x > 1)$ is a *contact suffix* if $T_{x-1} \in \mathcal{P}_j$ and $\mathcal{P}_j \notin A$. A subproblem $\mathcal{P}_i \in A$ is a *contact subproblem* if it contains at least one contact suffix. Each leaf of A is a contact node. Also, some internal nodes may be contact ones. When we consider the contact nodes in preorder (see grey nodes in Fig. 2), we call this the *contact visiting order*. Contacts nodes are useful for the refinement.

Status. An agglomerate A is *exhausted* if all its subproblems are either exhausted or solved. An unsolved subproblem of an agglomerate A is a *leading subproblem* if none of its ancestors in A are unsolved. An agglomerate A is *unsolved* if the following holds:

Property 1 (Leading Subproblem Property) *There exists a leading subproblem $\mathcal{P}_w$ in A s.t. (i) Each ancestor of $\mathcal{P}_w$ has only one child, and (ii) none of the ancestors of $\mathcal{P}_w$ is a contact node.*

Note that Property 1.(i) implies that an unsolved agglomerate has only one leading subproblem. At any time, a subproblem $\mathcal{P}_i \in A$ is *active* iff A is unsolved and either (i) $\mathcal{P}_i$ is unsolved or (ii) $\mathcal{P}_i$ is not solved and is a contact node of A (in this case $\mathcal{P}_i$ can be exhausted but still active). For any agglomerate A and two active subproblems $\mathcal{P}_i, \mathcal{P}_j \in A$, the following properties hold: (a) α_i and α_j have a non-void common suffix; (b) if $\mathcal{P}_i$ is a descendant of $\mathcal{P}_j$ in A then α_j is a proper suffix of α_i.

Columns. The agglomerates induce columns: a *column* C of A is the pair $\langle c_i, r_i\rangle$ such that $T_{c_i} \sqsubseteq^+ T_{r_i}$ where T_{c_i} is a contact suffix (i.e. it belongs to one of A's contact nodes) and T_{r_i} is a root suffix of A (i.e. it belongs to A's root). Each column $T[c_i, r_i]$ is a contiguous substring of T and, at any time, all the columns of all the agglomerates form a non-overlapping partition of T. A column $\langle c_i, r_i\rangle$ is associated with the contact subproblem $\mathcal{P}_j$ such that $T_{c_i} \in \mathcal{P}_j$. We will denote by $\|\mathcal{P}_j\|$ the number of columns associated with a contact subproblem $\mathcal{P}_j$. In Fig. 2, each agglomerate has its columns depicted beside its tree. The columns are shown as contiguous substrings of T (and T as a non-overlapping partitioning of the columns). For any agglomerate A, the number of its root suffixes, the number of its contact suffixes, the number of its columns and, if A is unsolved, the number of suffixes in its leading subproblem are all the same: we denote this quantity by $\|A\|$.

Data structures. The algorithmic challenge behind agglomerates is that the total cost of refining each individual subproblem could be prohibitive: there is a hidden rescanning cost that we must avoid. As we will describe late, the refining algorithm will pick an unsolved agglomerate A, and process it by refining *simultaneously* all of its subproblems according to the ranks of its leading subproblem $\mathcal{P}_w$. This is a non-trivial task. For example, to achieve optimality (Section 6) the refinement of A must be executed in time proportional to the number $|\mathcal{P}_w|$ (i.e $\|A\|$) of suffixes in $\mathcal{P}_w$ plus the number of newly created subproblems, whereas A can contain many more subproblems and suffixes.

The main structure for an agglomerate A is its Hasse diagram according to $\sqsubseteq^+$, which is a tree with $|A|$ nodes, each one representing a subproblem of A. Double links between children and parent are maintained. If A is unsolved, we also maintain (a) a pointer to its leading subproblem $\mathcal{P}_w$ and (b) the number of nodes in the path between A's root and $\mathcal{P}_w$. Only $2\|A\|$ suffixes are stored for A, i.e. its $\|A\|$ columns. They are divided into lists, one for each contact subproblem of A: the one for $\mathcal{P}_j$ contains all the $\|\mathcal{P}_j\|$ columns $\langle c_i, r_i\rangle$ such that $T_{c_i} \in \mathcal{P}_j$ (see Fig. 2).

We call *skip node* one that is a branching node (i.e. one with two or more children) or a contact node or the root of A (the conditions are not mutually exclusive). For each skip node $\mathcal{P}_j$ we maintain a *skip link* which is a double link between $\mathcal{P}_j$ and its lowest ancestor that is also a skip node. Clearly, the graph induced by the skip links is also a tree. We refer to it as A's *skip tree.* For each internal skip node $\mathcal{P}_j$ and each child $\mathcal{P}_{j_i}$ of $\mathcal{P}_j$ in the skip tree, we maintain a *guide link* that goes from $\mathcal{P}_j$ to the highest node of A that is (i) *unsolved* and (ii) both a descendant of $\mathcal{P}_j$ and an ancestor of $\mathcal{P}_{j_i}$ (if any such node exists). The skip tree of A with its guide links requires $O(\|A\|)$ space. In Fig. 2, skip and guide links are depicted as dashed arcs and dotted arrows.

Besides the pointers needed by the linked structures containing it, each subproblem $\mathcal{P}_i$ carries $O(1)$ words of information: (a) its integer label ℓ_i, (b) $|\mathcal{P}_i|$ and, when $\mathcal{P}_i$ is a contact node, the following additional information: (c) $\|\mathcal{P}_i\|$ (d) a pointer to the list of its columns and (e) a pointer to the root of A. For a generic subproblem $\mathcal{P}_i$, we do not explicitly maintain (e.g. in a list associated with $\mathcal{P}_i$) either its suffixes or its ranks in $\mathcal{R}(\mathcal{P}_i)$. The space required to store an agglomerate A is just $O(|A| + \|A\|)$ memory words.

We also maintain some *global* bookkeeping structures that are shared among the agglomerates. (a) For each contact or root suffix T_i, a pointer `Suff`$[i]$ to the subproblem to which it belongs. (b) A sorted linked list `SubList` of all the subproblems. (c) An array `Ranks` to find the nearest rank in $\mathcal{R}$ in constant time, so that the set of ranks $\mathcal{R}(\mathcal{P}_i)$ of any unsolved subproblem $\mathcal{P}_i$ can be retrieved in $O(|\mathcal{R}(\mathcal{P}_i)|)$ time.

3.3 Basic refining operations: slicing and joining agglomerates

Before describing the general refining scheme in Section 4, we discuss some useful operations that operate on agglomerates. The Slice operation takes in input an agglomerate A and assumes that each column $\langle c_j, r_j\rangle$ of A has been tagged with an integer in $\{1, \ldots, d\}$, for some $d \leq \|A\|$. For any suffix T_i of a subproblem in A, let us denote by $\tau(T_i)$ the tag of the column $\langle c_j, r_j\rangle$ to which it belongs ($c_j \leq i \leq r_j$). Slice(A) obtains the following *without* changing the columns of A:

(a) Each subproblem $\mathcal{P}_x \in A$ is partitioned (not necessarily this gives a refinement) into new subproblems $\mathcal{P}_{x_1}, \ldots, \mathcal{P}_{x_{d'}}$, where $d' \leq d$ and all the suffixes $T_i \in \mathcal{P}_{x_j}$ have the same tag $\tau(T_i)$, for each $1 \leq j \leq d'$ (after the partitioning $\mathcal{P}_x$ ceases to exist). If all the suffixes of some $\mathcal{P}_x \in A$ already have the same tag, $\mathcal{P}_x$ remains unmodified.

(b) All the subproblems in A, both the new ones and the unmodified ones, are distributed into new agglomerates $A_1, \ldots, A_d$ such that, for each $1 \leq t \leq d$ and for each suffix $T_i \in \mathcal{P}_{x_y} \in A_t$, we have that $\tau(T_i) = t$. Hence $\sum_{i=1}^{d} \|A_i\| = \|A\|$, with $\left|\bigcup_{i=1}^{d} A_i - A\right|$ newly created subproblems.

Lemma 1 *Slicing A into $A_1, \ldots, A_d$ takes $O\Big(\|A\| + \left|\bigcup_{i=1}^{d} A_i - A\right|\Big)$ time.*

The *join operation* Join(A', A) is much simpler. An agglomerate A' is *joinable* with A if there exists a contact subproblem $\mathcal{P}_x \in A$ such that, for each suffix T_i of the root subproblem $\mathcal{P}_{r'}$ of A', it is $T_{i+1} \in \mathcal{P}_x$. Hence $A' \cup A$ is an agglomerate and, by joining with A, we have that A' disappears and only A remains (their trees are fused with $\mathcal{P}_{r'}$ child of $\mathcal{P}_x$), thus some columns are *modified.*

Lemma 2 *If A' is joinable with A then the join operation requires $O(\|A'\|)$ time.*

During the computation we may need to combine the above two operations. Let agglomerates A and A_* be unsolved and exhausted, respectively. Also let us assume that A is joinable with A_* and let $\mathcal{P}_r$ be A's root. The SliceJoin(A, A_*) operation does the following: (i) it slices A_* into A_{*1} and A_{*2}, where $\langle c_j, r_j\rangle$ is a column of A_{*1} iff $T_{c_j - 1} \in \mathcal{P}_r \in A$; (ii) it joins A with A_{*1}.

Lemma 3 SliceJoin(A, A_*) *requires $O(\|A\| + |A_{*1}|)$ time.*

4 Optimal Algorithm for Multi-Selection of Suffixes

Initialization stage. We begin by initializing the bookkeeping data structures described in Section 3.2. Then, let MselMset be an optimal multi-selection algorithm for a *multiset* of items, thus generalizing the result in [7] to multisets (see Section 5.1). We call MselMset$(\{T[1], \ldots, T[N]\}, \mathcal{R})$ using the alphabet order: this refines the suffixes in S according to their first symbols into the pivotal multisets $\mathcal{M}_0$, $\mathcal{F}_1$,$\mathcal{M}_1$, ...,$\mathcal{F}_t$, $\mathcal{M}_t$. Each $\mathcal{M}_i \neq \emptyset$ forms a degenerate subproblem $\mathcal{P}_i$ since no ranks fall within it, whereas each $\mathcal{F}_j$ forms either a solved or unsolved subproblem $\mathcal{P}_j$. The agglomerates are initialized as singletons: each $\mathcal{P}_x$ forms an agglomerate A_x that is either exhausted (i.e. $\mathcal{P}_x$ is either solved or exhausted) or unsolved (i.e. it satisfies Property 1). Each A_x is moved into one of the groups `unsolved_g` and `exhausted_g`, which will be used in the rest of the computation.

Refinement stage. We proceed with the *refine and aggregate stage*. We execute a loop until the group `unsolved_g` is empty. In each iteration of the loop we pick an agglomerate from `unsolved_g` and we apply to it the *Refine and Aggregate Process*, shortly RAP, described in Section 4.1. During each RAP, some *temporary agglomerates* will be created and they may not fulfill the characterization given in Section 3.2. Thus, any agglomerate created during a RAP is to be considered as *temporary* until it disappears (since it is joined with another) or it is moved into either one of the global groups `unsolved_g` and `exhausted_g`, at the end of RAP.

Finalization stage. Once `unsolved_g` is empty, we run a finalization stage that returns the K wanted suffixes $T_{i_1} < T_{i_2} < \cdots < T_{i_K}$ by suitably scanning `SubList`.

4.1 The Refine and Aggregate Process (RAP)

First step: establishing which kind of agglomerate. The first kind is when A is *core cyclic*: there is one contact subproblem $\mathcal{P}_{x_*} \in A$, called the *core* of A, such that (a) there exists at least one column $\langle c_j, r_j \rangle$ of A with $T_{r_j+1} \in \mathcal{P}_{x_*}$, and (b) for each column $\langle c_i, r_i \rangle$ of A, either $T_{r_i+1} \in \mathcal{P}_{x_*}$ or $T_{r_i+1} \in \mathcal{P}_{y_i} \notin A$. For example, consider Fig. 2, where $A \equiv A_2$ is core cyclic with core $\mathcal{P}_{x_*} \equiv \mathcal{P}_{19}$: the only two columns $\langle c_i, r_i \rangle$ of A_2 with $T_{r_i+1} \in \mathcal{P}_{19}$ are $C_{21} = \langle 132, 133 \rangle$ and $C_{22} = \langle 134, 135 \rangle$, while the others have $T_{r_i+1} \in \mathcal{P}_{y_i} \notin A_2$. The columns in the core $\mathcal{P}_{x_*}$ are all *equal* and *consecutive* inside T. The second kind is when A is *generic* (i.e. not core cyclic) as illustrated by the example in Fig. 2: agglomerate A_3 is generic and "acyclic", as each column $\langle c_i, r_i \rangle$ of A_3 has $T_{r_i+1} \in \mathcal{P}_{y_i} \notin A_3$; instead, agglomerate A_1 is generic and "cyclic", as $C_1 = \langle 53, 60 \rangle$ and $C_{16} = \langle 118, 120 \rangle$ have $T_{61} \in \mathcal{P}_{17} \in A_1$ and $T_{121} \in \mathcal{P}_{16} \in A_1$. More formally, a generic and "acyclic" agglomerate A has each of its columns $\langle c_i, r_i \rangle$ with $T_{r_i+1} \in \mathcal{P}_{y_i} \notin A$; a generic and "cyclic" agglomerate A is one where there exist columns $\langle c_j, r_j \rangle$ and $\langle c_{j'}, r_{j'} \rangle$ such that $T_{r_j+1} \in \mathcal{P}_x \in A$ and $T_{r_{j'}+1} \in \mathcal{P}_{x'} \in A$ but $\mathcal{P}_x \neq \mathcal{P}_{x'}$. Note that cyclic agglomerates of the second kind are treated differently from those of the first kind.

Second step: building the keys for the agglomerate. We assign a *key* to each column of agglomerate A. If A is core cyclic (i.e. first kind), the key for each column $\langle c, r \rangle$ is as follows. Let $\langle c_1, r_1 \rangle, \langle c_2, r_2 \rangle, \ldots, \langle c_f, r_f \rangle$ be the maximal sequence of (equal) columns of A such that $r + 1 = c_1, r_1 + 1 = c_2, \ldots, r_{f-1} + 1 = c_f$. The key for $\langle c, r \rangle$ is the sequence $T_{c_1} T_{c_2} \ldots T_{c_f} T_{r_f+1}$. If A is generic (i.e. second kind, acyclic or not), each column $\langle c, r \rangle$ simply gets T_{r+1} as key. What about individual suffixes? The key of each T_i is *implicitly* the same as that assigned to *its* column $\langle c_j, r_j \rangle$, where $c_j \leq i \leq r_j$. However, unlike the columns of A, we do not access each suffix of A to actually assign it its key, which is retrieved on the fly only when needed. Recall that at any time the total order $\triangleleft$ and the equivalence relation $\simeq$ are defined on the suffixes (Section 3.1). Hence, any two keys can be compared in $O(1)$, since these suffixes are either contact or root, so we can use `Suff` to retrieve their labels: if A is core cyclic, its key becomes $\ell_i \ell_i \cdots \ell_i \ell_j$ for labels ℓ_i (repeated f times for T_{c_1}, T_{c_2}, $\ldots$, T_{c_f}) and ℓ_j (for T_{r_f+1}); if A is generic, its key becomes a single label (for T_{r+1}).

Third step: multi-selection on the leading subproblem seen as a multiset. We retrieve the ranks $r^w_1, \ldots, r^w_{|\mathcal{R}(\mathcal{P}_w)|}$ of the leading subproblem $\mathcal{P}_w$ of A, using `Ranks` (Section 3.2). We also retrieve all the suffixes of $\mathcal{P}_w$ (not explicitly stored for $\mathcal{P}_w$) and their keys computed in the second step, in $O(\|\mathcal{P}_w|) = O(\|A\|)$ time. Then, we call $\textsc{MselMset}\big(\mathcal{P}_w,\{\rho_1, \ldots, \rho_{|\mathcal{R}(\mathcal{P}_w)|}\}\big)$, where the ranks are $\rho_j = r^w_j - \ell_w$ for the label ℓ_w of $\mathcal{P}_w$, and the order among keys is the one given by $\triangleleft$ and $\simeq$. Let the resulting pivotal multisets be $\mathcal{M}_0, \mathcal{F}_1, \mathcal{M}_1, \ldots, \mathcal{F}_t, \mathcal{M}_t$, where $\mathcal{F}_j$ contains the suffixes that are candidates for some ranks and $t \leq |\mathcal{R}(\mathcal{P}_w)|$. We assign *tags* to the columns in A that are consistent with the order among these pivotal multisets, so that suffixes in the same multiset receive the same tag. (This is useful for the refinement in the next step.) For each $0 \leq j \leq t$, let h_j be the number of $\mathcal{M}_x \neq \emptyset$ with $0 \leq x \leq j$ (some of the $\mathcal{M}_j$'s may be empty). We use them to tag each column C of A: the tag is h_0 if the suffix of $\mathcal{P}_w$ that corresponds to C belongs to $\mathcal{M}_0$; the tag is $j + h_{j-1}$ if that suffix belongs to $\mathcal{F}_j$, or $j + h_j$ if that suffix belongs to $\mathcal{M}_j$, for some $1 \leq j \leq t$.

Fourth step: slicing the agglomerate. We perform the slicing of A (Section 3.3) using the d tags computed in the third step. We call $\textsc{Slice}(A)$ and obtain agglomerates $A_1, \ldots, A_d$, where the columns in each A_i have the same tag and, for $i \neq j$, the tag of A_i is different from that of A_j. Then $A_1, \ldots, A_d$ are moved into four groups: the two global ones previously mentioned, `exhausted_g` and `unsolved_g`, and two local ones for temporary agglomerates, called `undecided_g` and `joinable_g`, according to the following rule for the given A_i, where $1 \leq i \leq d$:

- `if` A_i's tag corresponds to a multiset $\mathcal{M}_j$ (see the third step), move it to `undecided_g`;
- `else` A_i's tag corresponds to a multiset $\mathcal{F}_j$:
  - `if` $\|A_i\| = 1$, move A_i to `exhausted_g`;
  - `else if` A was generic and there is a column $\langle c_i, r_i \rangle$ of A_i with $T_{r_i+1} \in \mathcal{P}_x \in A$, move A_i to `unsolved_g` (since A_i satisfies Property 1);
  - `else` move A_i to `joinable_g`.

This step has a subtle point, since it implicitly induces the refinement of many subproblems simultaneously without paying the rescanning cost. Not only the leading subproblem $\mathcal{P}_w$ is refined into its pivotal multisets using its ranks in $\mathcal{R}(\mathcal{P}_w)$, but each active subproblem $\mathcal{P}_i$ of A is refined as well. However, the refinement of $\mathcal{P}_i \neq \mathcal{P}_w$ could be coarser than what obtained by refining $\mathcal{P}_i$ directly using its ranks in $\mathcal{R}(\mathcal{P}_i)$: indeed, as pointed out in Section 3.2, the α-string α_w is a proper suffix of α_i, the α-string of $\mathcal{P}_i$ (descendant of $\mathcal{P}_w$). But $\mathcal{P}_i$'s induced refinement is for free since the cost is charged to $\mathcal{P}_w$, and any subsequent refinement for $\mathcal{P}_i$ will surely create new subproblems, for which we can pay (see what claimed for data structures in Section 3.2 and Lemma 1).

Fifth step: processing `undecided_g`. The group `undecided_g` collects the temporary agglomerates A_i's for which there are no ranks from $\mathcal{R}(\mathcal{P}_w)$ falling within A_i. However, there could be other ranks in $\mathcal{R} - \mathcal{R}(\mathcal{P}_w)$ that could involve A_i. Hence, (*i*) A_i may or may not contain unsolved subproblems, and (*ii*) even if A_i contains unsolved subproblems, it may not satisfy Property 1. (Here we may create a neighborhood from a subproblem of A_i as discussed in Section 3.1.) For each A_i in `undecided_g`, we retrieve the topmost unsolved subproblems $\mathcal{P}_{i_1} < \mathcal{P}_{i_2} < \cdots < \mathcal{P}_{i_{d_i-1}}$ from A_i in preorder, in $O(\|A_i\|)$ time using its skip tree (Section 3.2). Since no ancestor of $\mathcal{P}_{i_t}$ is unsolved, we assign tag t to its columns, $1 \leq t \leq d_i - 1$. We assign tag d_i to the remaining columns of A_i, which are not associated with any $\mathcal{P}_{i_t}$. We call $\textsc{Slice}(A_i)$ with these d_i tags: for $1 \leq t \leq d_i - 1$, we create a new agglomerate A_{i_t} with leading subproblem P_{i_t} and put it into `unsolved_g`. We create a new agglomerate $A_{i_{d_i}}$, exhausted by construction, and put it into `exhausted_g`.

Sixth step: processing `joinable_g`. This step represents the "aggregate" part of RAP. For each agglomerate A_i in `joinable_g`, let A_{i*} be the agglomerate with which A_i is joinable (Section 3.3). Note that A_{i*} is either in `unsolved_g` or `exhausted_g` (but not in `joinable_g`, see the fourth step).

If A_{i*} is in `unsolved_g`, or if $\|A_{i*}\| = \|A_i\|$, we call $\textsc{Join}(A_i, A_{i*})$ and move the resulting agglomerate in `unsolved_g`. In this way, we maintain Property 1: if A_{i*} is unsolved, then it satisfies the property; if A_{i*} is exhausted, the leading subproblem of A_i is also viable for the agglomerate obtained from $\textsc{Join}(A_i, A_{i*})$ (since $\|A_{i*}\| = \|A_i\|$).

If A_{i*} is in `exhausted_g` and $\|A_{i*}\| > \|A_i\|$: we call $\textsc{SliceJoin}(A_i, A_{i*})$ producing A_{i*1} and A_{i*2}. We move A_{i*1} to `unsolved_g` as it satisfies Property 1. We move A_{i*2} to `exhausted_g`: since A_{i*} was exhausted, it did not have a leading subproblem and A_i's leading subproblem is not viable for the resulting A_{i*2} (since $\|A_{i*}\| > \|A_i\|$, at least one of the conditions of Property 1 is violated).

5 Details of the Optimal Algorithm

5.1 Multi-selection on multisets

Consider the multi-selection problem on *multisets* of elements that are comparable in $O(1)$ time. The multi-selection algorithm in [7] does not exploit the presence of equal elements. Let us describe our variant. Given a multiset $\mathcal{M}$ and K ranks $r_1 < \cdots < r_K$, we want to partition $\mathcal{M}$ into its *pivotal multisets* $\mathcal{M}_0, \mathcal{F}_1, \mathcal{M}_1, \ldots, \mathcal{F}_t, \mathcal{M}_t$ such that the following holds:

(i) For any $0 < i < j \leq t$, for any $p_i \in \mathcal{F}_i, e_i \in \mathcal{M}_i, p_j \in \mathcal{F}_j, e_j \in \mathcal{M}_j$, we have that $p_i < e_i < p_j < e_j$; moreover, for any $e_0 \in \mathcal{M}_0, p_1 \in \mathcal{F}_1$, we have that $e_0 < p_1$.

(ii) The elements in $\mathcal{F}_i$ are equal and $|\mathcal{F}_i|$ is the multiplicity of $p_i \in \mathcal{F}_i$.

(iii) For each rank r_i there exists a $\mathcal{F}_j$ such that each p_j in $\mathcal{F}_j$ has rank r_i.

(iv) For each $\mathcal{F}_j$ there exists a rank r_i such that each p_j in $\mathcal{F}_j$ has rank r_i.

Notice that $t \leq K$ and $|\mathcal{F}_i| \geq 1$ whereas there may be some $\mathcal{M}_i = \emptyset$. Let us now describe our algorithm.

```
MselMset($\mathcal{M}, r_1, \ldots, r_K$)
1. If $K = 0$ exit. If $K = 1$ select and output the element of rank $r_1$.
2. Find the largest $r_l \leq \lceil \frac{N}{2} \rceil$. Then select the element $p'$ ($p''$) with rank $r_l$ ($r_{l+1}$) and all the elements of $\mathcal{M}$ that are equal to $p'$ ($p''$).
3. Partition $\mathcal{M}$ into $\mathcal{M}', \mathcal{F}', \mathcal{M}'', \mathcal{F}'', \mathcal{M}'''$ such that ($a$) $\mathcal{F}'$ ($\mathcal{F}''$) contains all the elements in $\mathcal{M}$ equal to $p'$ ($p''$) and ($b$) for any $e' \in \mathcal{M}', e'' \in \mathcal{M}'', e''' \in \mathcal{M}'''$ we have that $e' < p' < e'' < p'' < e'''$.
4. Find the largest $r_a \leq |\mathcal{M}'|$. Call MselMset($\mathcal{M}', r_1, \ldots, r_a$).
5. Let $q = |\mathcal{M}'| + |\mathcal{F}'| + |\mathcal{M}''|$. Find the smallest $r_b$ s.t. $r_b > q$.
   Call MselMset($\mathcal{F}'' \cup \mathcal{M}''', r_b - q, \ldots, r_K - q$).
```

The complexity of MselMset can be expressed in terms of the sizes of the pivotal multisets of $\mathcal{M}$ w.r.t. $r_1, \ldots, r_K$. For the sake of description, we will assume that $\log 0 = 0$.

Lemma 4 *The running time of the algorithm* MselMset *on a multiset $\mathcal{M}$ is upper bounded by $c|\mathcal{M}|\log|\mathcal{M}| - c|\mathcal{M}_0|\log|\mathcal{M}_0| - c\sum_{i=1}^{t}(|\mathcal{F}_i| + |\mathcal{M}_i|)\log(|\mathcal{F}_i| + |\mathcal{M}_i|) + c|\mathcal{M}|$ for a suitable integer constant c.*

Proof: From step 3 and because of the choice of p' and p'', it is clear that $\mathcal{F}'$,$\mathcal{M}''$ and $\mathcal{F}''$ are three of the pivotal multisets. Let $\mathcal{F}'$ and $\mathcal{M}''$ be $\mathcal{F}_s$ and $\mathcal{M}_s$, respectively (thus $\mathcal{F}''$ is $\mathcal{F}_{s+1}$). Also, the algorithm never breaks any pivotal multiset and the partitioning of the input elements for the recursive calls is also a partitioning of the (yet unknown) pivotal multisets. All the non-recursive steps of the algorithm require $\leq c|\mathcal{M}|$ time, for a suitable integer constant c. Therefore we have that the running time is upper bounded by the function $g(|\mathcal{M}|) = c|\mathcal{M}| + g(|\mathcal{M}'|) + g(|\mathcal{F}'' \cup \mathcal{M}'''|)$. To prove the upper bound for g we use the function $f(|\mathcal{M}|) = c|\mathcal{M}| + c(|\mathcal{F}_s| + |\mathcal{M}_s|)\log(|\mathcal{F}_s| + |\mathcal{M}_s|) + f(|\mathcal{M}'|) + f(|\mathcal{F}'' \cup \mathcal{M}'''|)$. Let us show that $f(|\mathcal{M}|) = g(|\mathcal{M}|) + c\sum_{i=1}^{t}(|\mathcal{F}_i| + |\mathcal{M}_i|)\log(|\mathcal{F}_i| + |\mathcal{M}_i|) + c|\mathcal{M}_0|\log|\mathcal{M}_0|$. Since the algorithm does not break $\mathcal{M}$'s pivotal multisets, we know that the pivotal multisets of $\mathcal{M}'$ and $\mathcal{F}'' \cup \mathcal{M}'''$ are $\mathcal{M}_0, \mathcal{F}_1, \mathcal{M}_1, \ldots, \mathcal{F}_{s-1}, \mathcal{M}_{s-1}$ and $\mathcal{F}_{s+1}, \mathcal{M}_{s+1}, \ldots, \mathcal{F}_t, \mathcal{M}_t$, respectively. Hence, by induction, we have that

$$f(|\mathcal{M}|) = c|\mathcal{M}| + c(|\mathcal{F}_s| + |\mathcal{M}_s|)\log(|\mathcal{F}_s| + |\mathcal{M}_s|) +$$

$$+g\left(|\mathcal{M}'|\right) + c\sum_{i=1}^{s-1}(|\mathcal{F}_i| + |\mathcal{M}_i|)\log(|\mathcal{F}_i| + |\mathcal{M}_i|) + c|\mathcal{M}_0|\log|\mathcal{M}_0| +$$

$$+g\left(|\mathcal{F}'' \cup \mathcal{M}'''|\right) + c \sum_{i=s+1}^{t} \left(|\mathcal{F}_i| + |\mathcal{M}_i|\right) \log\left(|\mathcal{F}_i| + |\mathcal{M}_i|\right).$$

Thus, by the definition of g, the relation between f and g is proven. All we have to do now is to prove that $f\left(|\mathcal{M}|\right) \leq c|\mathcal{M}|\log|\mathcal{M}| + c|\mathcal{M}|$, and the wanted upper bound for g will follow by subtraction. By the definition of $\mathcal{F}'$ we know that both $|\mathcal{M}'|$ and $|\mathcal{F}'' \cup \mathcal{M}'''|$ are less than $\frac{|\mathcal{M}|}{2}$. Thus, by induction we have the following: $f\left(|\mathcal{M}|\right) \leq c|\mathcal{M}| + c\left(|\mathcal{F}_s| + |\mathcal{M}_s|\right)\log\left(|\mathcal{F}_s| + |\mathcal{M}_s|\right) + c|\mathcal{M}'| + c|\mathcal{M}'|\log|\mathcal{M}'| + c|\mathcal{F}'' \cup \mathcal{M}'''| + c|\mathcal{F}'' \cup \mathcal{M}'''|\log|\mathcal{F}'' \cup \mathcal{M}'''| \leq c|\mathcal{M}| + c\left(|\mathcal{F}_s| + |\mathcal{M}_s|\right)\log|\mathcal{M}| + c|\mathcal{M}'| + c|\mathcal{M}'|\log\frac{|\mathcal{M}|}{2} + c|\mathcal{F}'' \cup \mathcal{M}'''| + c|\mathcal{F}'' \cup \mathcal{M}'''|\log\frac{|\mathcal{M}|}{2} = c|\mathcal{M}|\log|\mathcal{M}| + c|\mathcal{M}|$. ■

5.2 Dealing with the agglomerates in `undecided_g`

First: For each agglomerate A_i in `undecided_g`, we find its highest (i.e. closest to the tree root of A_i) unsolved subproblems (if any) and we collect them in $\mathtt{Lead}_i$. Since visiting the whole tree of A_i would cost too much ($O(|A_i|)$ time), we use A_i's skip tree and its guide links. In this way the visit takes $O(\|A_i\|)$ time. Specifically, we use two other lists $\mathtt{Con}_i$ and $\mathtt{F}_i$ (initially empty) besides $\mathtt{Lead}_i$, starting from the root with procedure LeadVisit($\mathcal{P}_r$) defined as follows for a generic subproblem $\mathcal{P}_r$:

1. If $\mathcal{P}_r$ is unsolved, we append $\mathcal{P}_r$, 1 and $|\mathcal{P}_r|$ at the end of $\mathtt{Lead}_i$, $\mathtt{F}_i$ and $\mathtt{Con}_i$, respectively, and return.

2. Otherwise, if $\mathcal{P}_r$ is a contact node, we append 0 and $\|\mathcal{P}_r\|$ to $\mathtt{F}_i$ and $\mathtt{Con}_i$, respectively (but we do not return yet).

3. For each child $\mathcal{P}_{r_j}$ of $\mathcal{P}_r$ (in A_i's skip tree), from the leftmost to the rightmost one, we do the following. If $\mathcal{P}_r$ has a guide link to an ancestor $\mathcal{P}_{r_x}$ of $\mathcal{P}_{r_j}$, we append $\mathcal{P}_{r_x}$, 1 and $|\mathcal{P}_{r_x}|$ at the end of $\mathtt{Lead}_i$, $\mathtt{F}_i$ and $\mathtt{Con}_i$, respectively, and return. Otherwise, if no such guide link exists, we call LeadVisit($\mathcal{P}_{r_j}$).

Second: For each agglomerate A_i in `undecided_g`, we tag its columns so that we are able to either (a) classify A_i as unsolved or exhausted or (b) partition A_i into some smaller unsolved or exhausted agglomerates. Basically, a column C of A_i has the tag l, $1 \leq l < |\mathtt{Lead}_i| + 1$, if the subtree rooted at the node $\mathcal{P}_j$ in position l in $\mathtt{Lead}_i$ contains the contact node with which C is associated. Otherwise, if no such node exists in $\mathtt{Lead}_i$, C has the tag $|\mathtt{Lead}_i| + 1$. Specifically, we compute the (inclusive) prefix sum $\mathtt{SCon}_i$ of $\mathtt{Con}_i$ and set $\mathtt{SCon}_i[0] = 0$. We scan the columns of A_i in contact visiting order (i.e. by using the preorder of the contact nodes in A_i). Let us consider the j-th one of them and let l, $1 \leq l \leq |\mathtt{SCon}_i|$, be the index such that $\mathtt{SCon}_i[l-1] < j \leq \mathtt{SCon}_i[l]$. The j-th column is tagged with l ($|\mathtt{Lead}_i| + 1$) if $\mathtt{F}_i[l] = 1$ ($= 0$).

Third: For each agglomerate A_i in `undecided_g`, we perform the slicing with the above tags. Let $t(i) = |\mathtt{Lead}_i|$. If the tags of A_i are all equal to some l s.t. $1 \leq l \leq t(i)$ (to $t(i) + 1$), we move A_i to `unsolved_g` (to `exhausted_g`) and the step ends. Otherwise, we call Slice(A_i) obtaining $A_{i_1}, \ldots, A_{i_{t(i)}}$ and, possibly, $A_{i_{t(i)+1}}$, where A_{i_l} is the agglomerate whose columns have l as tag, for each $1 \leq l \leq t(i) + 1$. The node in position l of $\mathtt{Lead}_i$ is the leading subproblem of A_{i_l}, for each $1 \leq l \leq t(i)$. Finally, we move $A_{i_{t(i)+1}}$ to `exhausted_g`, and all the other A_{i_j}'s to `unsolved_g`.

To understand why the above computation is correct, let us describe some properties of the tagging done. For any suffix T_j, let us denote with $\tau(T_j)$ the tag of the column $\langle c, r\rangle$ such that $c \leq j \leq r$.

If a subproblem $\mathcal{P}_r$ of A_i is *unsolved* then the following holds: (a) $\tau(T_{j'}) = \tau(T_{j''})$, for any two $T_{j'}, T_{j''} \in \mathcal{P}_r$ and (b) $1 \leq \tau(T_{j'}) \leq t(i)$.

On the other hand, if for a subproblem $\mathcal{P}_r$ of A_i we have that conditions (a) and (b) hold, then on the path from $\mathcal{P}_r$ to the root there has to be at least one *unsolved subproblem* whose suffixes have the same tag $\tau(*)$ as $\mathcal{P}_r$'s (maybe only $\mathcal{P}_r$ itself, if it is unsolved). Also, the highest unsolved subproblem on said path must be the one in position $\tau(T_{j'})$ in $\mathtt{Lead}_i$.

Given the definition of Slice in Section 3.3, the correctness follows.

5.3 Slicing agglomerates

As we have seen in Section 3.3, the slice operation receives in input an agglomerate A whose columns have been tagged with integers in $\{1, \ldots, d\}$, where $d \leq \|A\|$. Note that there should be at least two columns with different tags, since otherwise we do not need to run the slicing.

During the slice operation we deal with instances of the following *grouping problem*: We are given a list L of objects, each with an integer tag in $\{1, \ldots, d, d+1\}$ (where the tags of the columns are d but during parts of the slicing we will need an extra tag for special purposes). We want to partition L into $d' \leq d+1$ lists $L_{i_1}, \ldots, L_{i_{d'}}$ such that an object $o \in L_j$ iff o's tag is j, for each $j \in \{i_1, \ldots, i_{d'}\} \subseteq \{1, \ldots, d, d+1\}$. Note that the order in which the lists $L_{i_1}, \ldots, L_{i_{d'}}$ are produced does not necessarily have to follow the order of the tags. Also, the problem is easy if $|L| = \Omega(d)$ since it falls within the radix sort scheme. In our case, we can spend $O(d)$ preprocessing time and space beforehand: after that, for each instance L of the grouping problem, we assume that $|L| = o(d)$, and we cannot pay $O(d)$ time but just $O(|L|)$ time.

The procedure Group solves the above problem as follows. Between one call and the other, it reuses the same array J of $d+1$ slots that is allocated at the beginning of the slice operation and is never reset from one call to another. Each slot $J[i]$ has two fields: $J[i].p$, a list pointer, and $J[i].t$, an integer. Let η be an integer timestamp unique for each call (since we can just maintain an increasing integer throughout all the calls to Group). While we scan L, we build a list L' of lists and then return it. Let c be the tag of the current object o. During the scan we have two cases: (i) if $J[c].t \neq \eta$, we set $J[c].t = \eta$ and start a new list L_c with o as first element; also, we append L_c to L' and set $J[c].p$ to point to L_c; (ii) if $J[c].t = \eta$, we append o to the list pointed by $J[c].p$.

Lemma 5 *After $O(d)$ preprocessing time and space, each call to the procedure* Group *requires $O(|L|)$ time.*

At this point, we can describe Slice(A), which has three main phases: pruning, slicing, and finishing.

5.3.1 Pruning phase

The goal is to identify some relevant nodes in the tree reprensenting the agglomerate A. Speficically, a node $\mathcal{P}_i \in A$ is *homogeneous* if all the columns associated with all the contact nodes in the subtree rooted at $\mathcal{P}_i$ have the same tag. We want to find the set `Pruned` of the *highest* (i.e. closest to the root) *homogeneous nodes* of A and tag each one of them *with the common tag of their columns.* Let us recall (Lemma 1) that we need an implementation of Slice with a time complexity that is of the order of the number of columns of A plus the total number of the new subproblems created. Hence, we cannot touch all the homogeneous nodes of A since they will not be refined into new subproblems at this stage. If we are able to find the set `Pruned` efficiently then, later on, each subtree of A rooted at a node in `Pruned` will be just linked to the corresponding new agglomerate without accessing any of its internal nodes. The pruning phase proceeds with the following steps.

First: We traverse *the skip tree of* A level by level from the bottom one.

- For each $\mathcal{P}_l$ at the lowest level we do the following. (i) Since $\mathcal{P}_l$ is a contact node (and a leaf), we call Group on its (tagged) column list. (ii) Then we tag $\mathcal{P}_l$ with c (resp., with $d+1$) if from Group we get only one list L_c (resp., we get more than one list).
- For each $\mathcal{P}_i$ at a generic level (different from the lowest) we do the following. (i) If $\mathcal{P}_i$ is a leaf, we perform the same steps as we did at the lowest level. (ii) Otherwise, if $\mathcal{P}_i$ is an internal node, we call Group on the list of objects, where each object is one of $\mathcal{P}_i$'s children (and their tags); if $\mathcal{P}_i$ is also a contact node, we add more objects to the list, where each object is one of $\mathcal{P}_i$'s columns (which already have tags). (iii) Then we tag $\mathcal{P}_i$ as we did at the lowest level.

Second: We traverse A's *skip tree* with the following recursive SkipVisit($\mathcal{P}_r$), starting from the root. SkipVisit($\mathcal{P}_i$) is defined for a node $\mathcal{P}_i$ in the skip tree as follows: (i) if $\mathcal{P}_i$'s tag is $< d+1$, we output a pointer to $\mathcal{P}_i$ and return; (ii) otherwise, if the tag is $d+1$ we call SkipVisit($\mathcal{P}_{i_j}$), for each child (in the skip tree) $\mathcal{P}_{i_j}$ of $\mathcal{P}_i$.

Third: We are ready to retrieve and mark the nodes to be added to `Pruned`. (Note that the root $\mathcal{P}_r$ cannot belong to `Pruned`.) For each node $\mathcal{P}_i$ outputted in the previous step, we proceed as follows:

(i) We retrieve $\mathcal{P}_i$'s parent in the skip tree $\mathcal{P}_j$.

(ii) If $\mathcal{P}_i$ is the x-th child of $\mathcal{P}_j$ in the skip tree of A, we find $\mathcal{P}_j$'s x-th child in the tree of A, say $\mathcal{P}_{j_x}$: note that $\mathcal{P}_i$ is a descendant of $\mathcal{P}_{j_x}$ in the tree of A (they could be even the same node in some cases). Then we mark $\mathcal{P}_{j_x}$ (since we add it to `Pruned`) and tag it with $\mathcal{P}_i$'s tag.

(iii) If $\mathcal{P}_i$ and $\mathcal{P}_{j_x}$ are not the same node, we do the following. (a) We create a *temporary skip link* between them (for the next phase—the slicing phase). (b) Let v be the ancestor node of $\mathcal{P}_i$ pointed by the corresponding guide link of $\mathcal{P}_j$ (if any). For the sake of clarity, observe that traversing the tree of A from its root to $\mathcal{P}_i$, we meet $\mathcal{P}_j$, $\mathcal{P}_{j_x}$, v, and $\mathcal{P}_i$. We create a *temporary guide link* between $\mathcal{P}_{j_x}$ and v (they may be the same node in some cases).

Fourth: Let $\mathcal{Y}$ be the set of nodes of A that are not in a subtree rooted at a node belonging to `Pruned`. We assign to each $\mathcal{P}_i \in \mathcal{Y}$ a *fingerprint* that is a unique integer from $\{1,\ldots,|\mathcal{Y}|\}$ (any choice would do, for example the DFS numbering).

Lemma 6 *The pruning phase requires* $O\Big(\|A\| + \Big|\bigcup_{i=1}^{d} A_i - A\Big|\Big)$ *time.*

5.3.2 Slicing phase

The goal is to actually slice the agglomerate A into the agglomerates defined by the columns' tags, as claimed in Lemma 1, with some provisions. Indeed, the only missing things to complete this task are the following: (a) the link between each contact node and the root; (b) the contact node list; (c) the correct labels and the correct ordering in `SubList` for the new subproblems created; (d) the guide links of skip nodes that are not descendant of any node in `Pruned`. We will deal with these things in the next phase—the finishing phase.

We invoke the recursive procedure SliceRec() on the root of A. A generic call SliceRec($\mathcal{P}_r$), where $\mathcal{P}_r$ indcates now a generic node in A, has three cases.

$\mathcal{P}_r$ is a leaf. First, we call Group on $\mathcal{P}_r$'s column list and obtain $d' \leq d+1$ lists $L_{r_1}, \ldots, L_{r'_d}$. By scanning each L_{r_i} we obtain (a) all the distinct tags $t_1, \ldots, t_{d'}$ and (b) $n_1, \ldots, n_{d'}$ where n_i is the number of columns in list L_{r_i}.

Second, we create d' new subproblems $\mathcal{P}_{r_1}, \ldots \mathcal{P}_{r_{d'}}$ and insert them into `SubList` in place of $\mathcal{P}_r$. If $\mathcal{P}_r$ was A's leading subproblem, we set $\mathcal{P}_{r_1}, \ldots \mathcal{P}_{r_{d'}}$ to be leading subproblems of their respective agglomerates (they might not actually be, see Section 5.2). For each $1 \leq i \leq d'$: (a) we set $\mathcal{P}_{r_i}$'s tag and fingerprint to t_i and $\mathcal{P}_r$'s fingerprint, respectively; (b) we set $|\mathcal{P}_{r_i}|$, $\mathcal{P}_{r_i}$'s column list pointer and ℓ_{r_i} to n_i, L_{r_i} and ℓ_r, respectively. Finally, we eliminate $\mathcal{P}_r$ and return $\mathcal{P}_{r_1}, \mathcal{P}_{r_2} \ldots \mathcal{P}_{r_{d'}}$.

$\mathcal{P}_r$ is a contact node but not a leaf. If $\mathcal{P}_r \in$ `Pruned`, we return it immediately (it has been tagged in the previous phase–the pruning phase). Otherwise we proceed as follows.

First, we call SliceRec($\mathcal{P}_{r_i}$), for each child $\mathcal{P}_{r_i}$ of $\mathcal{P}_r$. From each call SliceRec($\mathcal{P}_{r_i}$) we receive a set $\mathcal{Q}_i$ of root nodes. We call Group on the list with the objects in $\mathcal{C} \cup \mathcal{Q}_1 \cup \cdots \cup \mathcal{Q}_x$, where $\mathcal{C}$ contains $\mathcal{P}_r$'s columns. This produces $d' \leq d+1$ lists $L_{r_1}, \ldots, L_{r'_d}$. By scanning the lists, we obtain (a) all the distinct tags $t_1, \ldots, t_{d'}$ and (b) pairs $\langle col_1, nod_1 \rangle, \ldots, \langle col_{d'}, nod_{d'} \rangle$, where lists col_i and nod_i contain all the columns and all the nodes in L_{r_i}, respectively (some of them may be empty). Then by scanning each col_i and nod_i we obtain (c) $n_1, \ldots, n_{d'}$, where n_i is the number of columns in col_i, and (d) $p_1, \ldots, p_{d'}$, where p_i is the total number of suffixes in each subproblem in list nod_i.

Second, we create d' new subproblems $\mathcal{P}_{r_1}, \ldots \mathcal{P}_{r_{d'}}$ and insert them in `SubList` in place of $\mathcal{P}_r$. If $\mathcal{P}_r$ was A's leading subproblem, we set $\mathcal{P}_{r_1}, \ldots \mathcal{P}_{r_{d'}}$ to be leading subproblems of their respective agglomerates (same as the above case when $\mathcal{P}_r$ is a leaf). For each $1 \leq i \leq d'$: (a) we set $\mathcal{P}_{r_i}$'s tag and fingerprint to t_i and $\mathcal{P}_r$'s fingerprint, respectively; (b) we set $|\mathcal{P}_{r_i}|$, $\mathcal{P}_{r_i}$ column list pointer (in case $\mathcal{P}_{r_i}$ is a new contact node) and ℓ_{r_i} to $n_i + p_i$, col_i and ℓ_r, respectively; (c) we make the nodes in nod_i be $\mathcal{P}_{r_i}$'s children.

Third, for each $\mathcal{P}_{r_i}$ and each child $\mathcal{P}_{r_{ij}}$ of $\mathcal{P}_{r_i}$ we do as follows.

(i) If $\mathcal{P}_{r_i}$ is not a skip node but its only child $\mathcal{P}_{r_{ij}}$ *is*, we create a temporary link between them.

(ii) If neither $\mathcal{P}_{r_i}$ nor $\mathcal{P}_{r_{ij}}$ is a skip node, we redirect to $\mathcal{P}_{r_i}$ the (only) temporary link that goes into $\mathcal{P}_{r_{ij}}$.

(iii) If $\mathcal{P}_{r_i}$ is a skip node and $\mathcal{P}_{r_{ij}}$ *is not*, we redirect to $\mathcal{P}_{r_i}$ the temporary link that goes into $\mathcal{P}_{r_{ij}}$, and we change it into a skip link.

(iv) If *both* $\mathcal{P}_{r_i}$ and $\mathcal{P}_{r_{ij}}$ are skip nodes, we create a skip link between them.

After that, we eliminate $\mathcal{P}_r$ and return $\mathcal{P}_{r_1}, \mathcal{P}_{r_2} \ldots \mathcal{P}_{r_{d'}}$.

$\mathcal{P}_r$ is not a contact node. This case is analogous to the previous one, only simpler, because $\mathcal{P}_r$ is not a contact node and no new contact nodes can be created from it.

Lemma 7 *The slicing phase requires* $O\Big(\|A\| + \Big|\bigcup_{i=1}^{d} A_i - A\Big|\Big)$ *time.*

5.3.3 Finishing phase

The finishing phase adds the missing information from the previous phase—the slicing phase. It proceeds with the following steps.

First: We sort *all* the new subproblems $\mathcal{P}_j$'s according the keys $\langle m_j, t_j \rangle$, where m_j and t_j are $\mathcal{P}_j$'s fingerprint and tag, respectively. Since each $\langle m_j, t_j \rangle$ has $O(\log|A|)$ bits, we can use radix sort.

Second: After the sorting, for each old *active* subproblem $\mathcal{P}_i$, we have that new subproblems $\mathcal{P}_{i_1}, \dots, \mathcal{P}_{i_x}$ are grouped together *and* in their correct relative order. Hence, we can reattach them in `SubList` in the correct order and also set the correct label ℓ_{i_j} for each one of them. If $\mathcal{P}_i$ was inactive, we leave $\mathcal{P}_{i_1}, \dots, \mathcal{P}_{i_x}$ and their labels as they are.

Third: For each new agglomerate A_i we build its contact node list by visiting its skip tree. Then we scan its contact node list and for each contact node we set the link to A_i's root.

Fourth: For each new agglomerate A_i we need to create the guide links for the skip nodes of A_i that are not descendants of the nodes in `Pruned`. To that end, we call GuideVisit($\mathcal{P}_i$), defined as follows, on A_i's root.

(i) For each child $\mathcal{P}_{i_j}$ of $\mathcal{P}_i$ in A's *skip tree*, we scan the nodes $\mathcal{P}_{i_x}$ of A's *tree* that are both *descendants of* $\mathcal{P}_i$ and *ancestors of* $\mathcal{P}_{i_j}$ starting from the highest one. We keep scanning them until we find a $\mathcal{P}_{i_x}$ that falls into one of three cases: (a) it is unsolved, (b) it is in `Pruned` or (c) it is $\mathcal{P}_{i_j}$. In case (a) we create a guide link between $\mathcal{P}_i$ and $\mathcal{P}_{i_x}$. In case (b) if $\mathcal{P}_{i_x}$ has a temporary guide link (possibly created in the pruning phase) to a node v, we create a guide link between $\mathcal{P}_i$ and v. Otherwise no guide link is created.

(ii) We call GuideVisit($\mathcal{P}_{i_j}$), for each child $\mathcal{P}_{i_j}$ of $\mathcal{P}_i$ in A's *skip tree.*

Lemma 8 *The finishing phase requires* $O\Big(\|A\| + \Big|\bigcup_{i=1}^{d} A_i - A\Big|\Big)$ *time.*

5.4 Joining agglomerates

Given an agglomerate A' that is joinable to A, we need to attach the root of A' to a suitable place in A. Let $\mathcal{P}_x \in A$ be the contact subproblem such that $T_{i+1} \in \mathcal{P}_x$, for each suffix T_i of the root subproblem $\mathcal{P}_{r'}$ of A'. The operation Join(A', A) (see Section 3.3) proceeds with the following steps.

First: We fuse the trees of A and A' by making $\mathcal{P}_{r'}$ be the new leftmost children of $\mathcal{P}_x$.

Second: Let L and L' be the contact node lists of A and A', respectively. Let p and s be the predecessor and successor of $\mathcal{P}_x$ in L, respectively. Let b and e be the leftmost and the rightmost node in L', respectively. If after the fusion $\mathcal{P}_x$ is not (is still) a contact node, we link p ($\mathcal{P}_x$) to b and e to s.

Third: Let us define v as follows: (i) if $\mathcal{P}_x$ is still a skip node after the fusion (it may not be a contact node anymore), then v is $\mathcal{P}_x$; (ii) otherwise v is the ancestor of $\mathcal{P}_x$ pointed by its skip link. If $\mathcal{P}_{r'}$ is a contact or branching node, we create a skip link between v and $\mathcal{P}_{r'}$. Otherwise $\mathcal{P}_{r'}$ is not a skip node of the final agglomerate. Hence, the only skip link from a node in A' to $\mathcal{P}_{r'}$ is redirected to v. We do the same with the only guide link of $\mathcal{P}_{r'}$.

Fourth: Each column $\langle c'_i, r'_i \rangle$ of A' is changed into $\langle c'_i, r_j \rangle$ where $\langle c_j, r_j \rangle$ is the column of A associated with $\mathcal{P}_x$ such that $r'_i + 1 = c_j$. The column $\langle c_j, r_j \rangle$ is deleted since T_{c_j} is not a contact suffix anymore.

Fifth: We set the root pointer of each contact node of A' to A's root.

Since the total number of skip links, columns and contact nodes of A' and deleted pairs of A is $O(\|A'\|)$, Lemma 2 is proven.

5.5 Slicing and joining with an exhausted agglomerate

Let us now describe the SliceJoin(A, A_*) operation used in the last step of rap and whose effect has been described in Section 3.3. Let A and A_* be unsolved and exhausted, respectively, and let us assume that A is joinable with A_*. Let $\mathcal{P}_x$ be the contact node of A_* such that $T_{i+1} \in \mathcal{P}_x$, for each suffix T_i of the root subproblem $\mathcal{P}_r$ of A.

For the slicing part of SliceJoin, we *do not* explicitly tag A_*'s columns but, conceptually, we would have the following: (i) only two tags (1 and 2) for the columns; (ii) $\mathcal{P}_x$ is the only contact subproblem of A_* with some columns with tag 1, and all the other columns with tag 2. Because of that, we know that the only subproblems of A_* that are partitioned during the slicing part of SliceJoin are $\mathcal{P}_x$ and its ancestors. Thus, we do not need the pruning phase of Slice because $\mathcal{P}_x$ is the only non-homogeneous node.

The slicing phase is the same as in Slice except for two things. First, the recursion does not touch any node that is not on the path from the root of A_* to $\mathcal{P}_x$ (since they are implicitly in `Pruned`). Second, when we treat $\mathcal{P}_x$ (which is a contact node) we do not touch its column list at all, we just create the two subproblems $\mathcal{P}_{x_1}$ and $\mathcal{P}_{x_2}$, and we link *the whole column list* of $\mathcal{P}_x$ to $\mathcal{P}_{x_2}$ (which will be part of A_{*2} and is still a contact node). We can leave in the column list of $\mathcal{P}_{x_2}$ all those columns that, after a normal Slice of A_*, would end up being associated with $\mathcal{P}_{x_1}$ without incurring in any trouble for two reasons: (i) after the join of A and A_{*1} they would disappear anyway, and (ii) A_{*2} is exhausted and its contact nodes are not active anymore.

We also do not need the finishing phase of Slice, since all the subproblems in A_* are exhausted and A will join with A_{*1} after the slicing part of SliceJoin.

Because of the characteristics of A_{*1}, to join A with it we just (i) link A's root to the only contact node of A_{*1} and (ii) change each column $\langle c_j, r_j \rangle$ of A to $\langle c_j, r_j + l \rangle$, where l is the number of nodes A_{*1} (whose tree is just a path).

Thus, Lemma 3 follows as a corollary of Lemma 1.

5.6 Finalization stage

We finally have to store into `SubList` all the K wanted suffixes. Note that we need to retrieve them from the columns that contains them. To this end, we output the set of K pairs `RkSuff` $= \{\langle r_i, j \rangle \,| r_i \in \mathcal{R}$ and T_j has rank $r_i\}$ in the following way. We scan `SubList` and for each subproblem $\mathcal{P}_x$ that is both *solved* and a *leaf* we do as follows. First we add $\langle r', j_x \rangle$ to `RkSuff`, where r' and T_{j_x} are $\mathcal{P}_x$'s only rank and only suffix. Then we retrieve all the ancestors of $\mathcal{P}_x$ (in the tree of its agglomerate) that are also solved. They are all the nodes closest to $\mathcal{P}_x$ in its leaf-to-root path. Let $\mathcal{P}_{x_y}$ be the y-th closest one of them, we add $\langle r^y, j_x + y \rangle$ to `RkSuff`, where r^y is the only rank of $\mathcal{P}_{x_y}$ (and T_{j_x+y} is clearly $\mathcal{P}_{x_y}$'s only suffix).

Lemma 9 *The finalization stage requires $O(N)$ time.*

6 Correctness and Analysis

Correctness and complexity are strictly related, so we discuss them together. here we give the lemmas needed to prove the theorems stated in the Introduction, and few proofs. We devote Section 7 to the remaining proofs.

Consider first a simplified scenario where we have to perform multi-selection on a prefix-free set $\mathcal{Z}$ of N *independent* strings of total length L, using our set $\mathcal{R}$ of K ranks. We adopt the same notation as in formula (1) and Section 3.

We run MselMset$(\mathcal{Z}, \mathcal{R})$ on the first symbol of all the strings in $\mathcal{Z}$. This partitions the strings into unsolved, solved and exhausted subproblems: $\mathcal{P}_i$ is unsolved when it contains all the strings with the same first symbols and $|\mathcal{P}_i| > |\mathcal{R}(\mathcal{P}_i)| \geq 1$; or, $\mathcal{P}_i$ is solved when $|\mathcal{P}_i| = |\mathcal{R}(\mathcal{P}_i)| = 1$; finally, if $\mathcal{P}_i$ is exhausted then $|\mathcal{R}(\mathcal{P}_i)| = 0$ and the first symbols of its strings may not be the same. We repeat the refining steps until there are no more unsolved subproblems. We pick any unsolved subproblem $\mathcal{P}_i$. Let a_i be the length of the common prefix (of its strings) examined so far: we invoke MSELMSET($\mathcal{P}_i, \mathcal{R}(\mathcal{P}_i)$) using the alphabetic order on the symbols $y[a_i + 1]$ for $y \in \mathcal{P}_i$. Thus we refine $\mathcal{P}_i$ into smaller subproblems, classify them as described above, and repeat the steps.

Lemma 10 *The running time of the multi-selection algorithm with K ranks for a prefix-free set of N independent strings of total length L is upper bounded by $O\left(N \log N - \sum_{j=0}^{K} \Delta_j \log \Delta_j + N + L\right)$.*

Here we focus on the multi-selection for our set S of N suffixes, and show how to consider them as a set of N independent *virtual* strings.

Lemma 11 *The running time of the suffix multi-selection algorithm for a text of length N is upper bounded by $O\left(N \log N - \sum_{j=0}^{K} \Delta_j \log \Delta_j + N + rap\right)$, where rap is the total time required by all the* RAP*s minus the time for the* MSELMSET *calls.*

Proof: Consider the computation described in Sections 3 and 4, and the *virtual symbol cost* of the calls to MSELMSET: when applied to a subproblem $\mathcal{P}_w$, it performs comparisons using a certain order on $\mathcal{P}_w$'s keys, which can be seen as the *virtual* symbols of independent strings. Indeed, these virtual symbols are exclusively created and "used" for $\mathcal{P}_w$. Unlike T's symbols, virtual symbols are not shared by subproblems. Each $\mathcal{P}_w$ has associated $|\mathcal{P}_w|$ virtual strings that are made up of all the virtual symbols *created* to refine $\mathcal{P}_w$ during several RAPs, every time $\mathcal{P}_w$ is the leading subproblem of its current agglomerate. And, unlike the suffixes of T, all the virtual strings are independent.

Let L be the total number of virtual symbols thus created by all the RAPs. Lemma 10 reports the virtual symbol cost for all the MSELMSET calls, that is, their total contribution to the final cost of our multi-selection algorithm. We have to add the cost of the rest of the computation, which is $O(rap)$ by definition. It remains to show that $L = O(rap)$, thus proving the claimed bound.

When MSELMSET is applied to $\mathcal{P}_w$, let A be the agglomerate for which $\mathcal{P}_w$ is its leading subproblem. The number $|\mathcal{P}_w|$ of virtual symbols created in this call satisfies $|\mathcal{P}_w| = \|A\|$. Since the RAP computation time for this step (minus the call to MSELMSET) is $\Omega(\|A\|)$ (e.g. Lemmas 1– 3), this computation time is an upper bound for $|\mathcal{P}_w|$. Summing up over all the RAPs, we obtain that the total number L of virtual symbols thus created is upper bounded by the computation time of all the RAPs minus the MSELMSET calls, namely, $L = O(rap)$. ■

Lemma 12 *The running time of the suffix multi-selection algorithm for a text of length N is upper bounded by $O(K \log K + N + rap)$ when $\mathcal{R}$ is an interval of K consecutive ranks.*

We need an intermediate stage to find the suffixes of ranks r_1 and r_K, and create a subproblem with the remaining $K - 2$ ones. After that, we run our multi-selection.

Lemma 13 *Independently of the choice of the ranks in $\mathcal{R}$, $rap = O(N)$.*

Proof: We give an analysis based on counting the following *types of events*: (a) *Subproblem creation*: when some $\mathcal{P}_j$ is partitioned into $\mathcal{P}_{i_1}, \ldots, \mathcal{P}_{i_p}$ (during a SLICE or at the beginning of a SLICEJOIN). (b) *Suffix discovery*: when some $T_w \in \mathcal{P}_w$ is recognized as one of the wanted suffixes with rank in $\mathcal{R}$ (third step of RAP). (c) *Suffix exhaustion*: when some suffix $T_e \in \mathcal{P}_w \in A$ becomes exhausted, where $\mathcal{P}_w$ is the leading subproblem of A (third step of RAP). (d) *Column fusion*: when a column of A' is fused with one of A (during JOIN(A', A) or at the end of a SLICEJOIN). (e) *Inner collision*: when processing a column $\langle c_i, r_i \rangle$ of A such that $\langle c_j, r_j \rangle$ is also in A and satisfies $c_j = r_i + 1$, while T_{c_i} and T_{c_j} do not belong to the same subproblem (second step of RAP).

Claim: There are overall $O(N)$ events occurring in any execution of our algorithms. Indeed, the same event cannot repeat. A column is never divided and a subproblem is never merged with others. If a suffix is exhausted or a wanted suffix is found, they can never be in an unsolved subproblem again. An inner collision is unique, since the two colliding columns will not be part of the same agglomerate. Since we start with N columns and one subproblem, the total number of events is $O(N)$.

For a generic RAP on an agglomerate A, let N_A be the number of events thus occurring. Since the overall number of events is $O(N)$, this implies that $\sum_A N_A = O(N)$. We show that $N_A \geq \|A\| + \Pi_A$, where Π_A is the number of created subproblems. Consider events (a), and observe that their contribution is Π_A, which is the sum of three quantities: $|\cup_{i=1}^{d} A_i - A|$, where $A_1, \ldots, A_d$ are the agglomerates into which A is sliced in the fourth step; $\sum_{\{A_i \in \texttt{undecided\_g}\}} |\cup_{j=1}^{d_i} A_{i_j} - A_i|$, where $A_{i_1}, \ldots, A_{i_{d_i}}$ are the agglomerates into which each A_i is sliced in the fifth step; $\sum_{\{A_i \in \texttt{slicejoinable\_g}\}} |A_{i*1}|$, where `slicejoinable_g` denotes all the A_i's in `joinable_g` that need a SLICEJOIN in the sixth step.

As for events (b)–(e), they totalize at least $\|A\|$ in number and occur in the fourth step. Namely, the number of (b)s and (c)s is at least the total number of columns of A_i's that go to `exhausted_g` or to `undecided_g`: each column of each A_i moved to `undecided_g` corresponds to (c), and each column of each A_i moved to `exhausted_g` corresponds to either (b) or (c). The number of (d)s and (e)s is at least the total number of columns of the A_i's that are moved to `joinable_g` and `unsolved_g`, respectively. Since the total number of involved columns in the fourth step is $\sum_{i=1}^{d} \|A_i\| = \|A\|$, we obtain the claimed number.

At this point, to prove $\mathit{rap} = O(N)$, it remains to see that the cost of a generic RAP (MSELMSET excluded) on an agglomerate A is $O(\|A\| + \Pi_A)$ time. The first three steps of the RAP take $O(\|A\|)$ time. The costs of the fourth and fifth steps are given by Lemma 1: precisely, $O\big(\|A\| + |\cup_{i=1}^{d} A_i - A|\big)$ and $O(\sum_{\{A_i \in \texttt{undecided\_g}\}} (\|A_i\| + |\cup_{j=1}^{d_i} A_{i_j} - A_i|))$. By Lemmas 2 and 3, the total cost of the sixth step is $O(\sum_{\{A_i \in \texttt{joinable\_g}\}} \|A_i\| + \sum_{\{A_i \in \texttt{slicejoinable\_g}\}} |A_{i*1}|)$. Since $\sum_{\{A_i \in \texttt{undecided\_g}\}} \|A_i\| + \sum_{\{A_i \in \texttt{joinable\_g}\}} \|A_i\| \leq \|A\|$, the total cost is $O(\|A\| + \Pi_A)$ and $\mathit{rap} = O(\sum_A (\|A\| + \Pi_A)) = O(\sum_A N_A) = O(N)$. ∎

For any two subproblems $\mathcal{P}_i$ and $\mathcal{P}_{i'}$ such that $\mathcal{N}_i \neq \mathcal{N}_{i'}$, let $\mathit{lcp}(\mathcal{P}_i, \mathcal{P}_{i'})$ be the length of the longest common prefix of any two suffixes $T_{j_i} \in \mathcal{P}_i$ and $T_{j_{i'}} \in \mathcal{P}_{i'}$. We have the following:

Lemma 14 *The suffix multi-selection algorithm can support* $\mathit{lcp}(\mathcal{P}_i, \mathcal{P}_{i'})$ *queries in* $O(1)$ *time, for any two* $\mathcal{P}_i, \mathcal{P}_{i'}$ *such that* $\mathcal{N}_i \neq \mathcal{N}_{i'}$*, without changing its asymptotic time complexity.*

Proof: Consider a snapshot of the computation, recalling that we maintain the sorted linked list `SubList` of all the subproblems (where some of them belong to the same neighborhood). Suppose by induction that we have the *lcp*'s between consecutive subproblems. We use a Cartesian Tree (CT) plus lowest common ancestor (LCA) dynamic queries to compute the *lcp* for any two subproblems. We maintain the induction at the end of the SLICE operation: when A_i is sliced into $A_{i_1}, \ldots, A_{i_d}$, the newly created subproblems $\mathcal{P}_{x_1}, \ldots, \mathcal{P}_{x_{d'}}$ for each A_{i_j} are stored contiguously in `SubList`. We compute $\mathit{lcp}(\mathcal{P}_{x_1}, \mathcal{P}_{x_2})$, $\ldots$, $\mathit{lcp}(\mathcal{P}_{x_{d'-1}}, \mathcal{P}_{x_{d'}})$ using using their keys, the CT, and the LCA queries, in $O(d')$ time, which we can pay for. Since these *lcp*'s are longer, we just need to add $d' - 1$ new leaves to CT, without increasing the asymptotic complexity. ∎

7 Proofs

7.1 Multi-selection on a set of strings

In order to prove the upper bound for the time complexity of our suffix multi-selection algorithm, we first need to prove the complexity of the following algorithm for multi-selection of *independent strings* (i.e. not sharing symbols).

We have a set $\mathcal{Z}$ of N *independent strings* of total length L such that none of them is the prefix of another. We want to select the K strings with ranks in $\mathcal{R} = \{r_1, \ldots, r_K\}$, where $r_1 < \ldots < r_K$. Let r_0 and r_{K+1} be 0 and $N+1$ respectively.

In our multi-selection algorithm for independent strings a subproblem $\mathcal{P}_i$ is a subset of $\mathcal{Z}$ associated with (*a*) $\mathcal{R}(\mathcal{P}_i) \subseteq \mathcal{R}$, (*b*) $less_i$, the number of strings in $\mathcal{Z}$ that are less than each one in $\mathcal{P}_i$ and (*c*) an offset $o\!f\!f_i \in \{1, \ldots, l_i\}$, where l_i is the length of the shortest string in $\mathcal{P}_i$.

1. We put in `unsolved_g` the first subproblem $\mathcal{P}_0 = \mathcal{Z}$ where $\mathcal{R}(\mathcal{P}_0) = \mathcal{R}$, $less_0 = 0$ and $o\!f\!f_0 = 1$.

2. We repeat the following steps until `unsolved_g` is empty.

   (a) We pick a subproblem $\mathcal{P}_i$ in `unsolved_g` and we call MselMset$(\mathcal{M}, \mathcal{R}_i^-)$ where $\mathcal{M} = \{s[o\!f\!f_i] \mid s \in \mathcal{P}_i\}$ and $\mathcal{R}_i^- = \{r_j - less_i \mid r_j \in \mathcal{R}(\mathcal{P}_i)\}$. This partitions $\mathcal{M}$ into its pivotal multisets $\mathcal{M}_0, \mathcal{F}_1, \mathcal{M}_1, \ldots, \mathcal{F}_t, \mathcal{M}_t$.

   (b) For each $\mathcal{F}_j$, we collect the set $\mathcal{P}_{i_j} \subseteq \mathcal{P}_i$ of the strings corresponding to the symbols in $\mathcal{F}_j$, and we set $o\!f\!f_{i_j} = o\!f\!f_i + 1$, $less_{i_j} = |\mathcal{M}_0| + \sum_{x=1}^{j-1} |\mathcal{F}_x \cup \mathcal{M}_x|$ and $\mathcal{R}(\mathcal{P}_{i_j}) = \{r_y \in \mathcal{R}(\mathcal{P}_i) \mid less_{i_j} < r_y \leq less_{i_j} + |\mathcal{P}_{i_j}|\}$.

   (c) If $|\mathcal{P}_{i_j}| = 1$ we move $\mathcal{P}_{i_j}$ to `sol_g`. Otherwise, we move $\mathcal{P}_{i_j}$ to `unsolved_g`.

3. For each $\mathcal{P}_i$ in `sol_g`, we output $\langle s_i, r_i \rangle$, where $s_i \in \mathcal{P}_i$ and $r_i \in \mathcal{R}(\mathcal{P}_i)$.

Lemma 15 *The running time of the multi-selection algorithm for a set $\mathcal{Z}$ of independent (and prefix free) strings of total length L is upper bounded by*

$$c\left(N \log N - \sum_{j=0}^{K} \Delta_j \log \Delta_j + N + 5L'\right)$$

where c is a suitable integer constant, $r_0 = 0$, $r_{K+1} = N+1$, $\Delta_j \equiv r_{j+1} - r_j$ for $0 \leq j \leq K$, and $L' \leq L$ is the smallest number of symbols we need to probe to find the wanted strings.

Proof: The *offset size* of a subproblem $\mathcal{P}_i$ is the total length of the strings in $\{s[o\!f\!f_i \cdots |s|] \mid s \in \mathcal{P}_i\}$. It is easy to see that the computations on distinct subproblems proceed independently from one another. Hence, we prove the thesis by induction on the offset size of subproblems.

Let us consider subproblem $\mathcal{P}_0$ (step 1), its corresponding multiset $\mathcal{M}$ and the pivotal multisets of $\mathcal{M}$ computed in step 2. Because of Lemma 4, after the first execution of step 2a the terms $cN \log N$ and cN of the wanted upper bound are accounted for.

Let us first deal with the pivotal multisets $\mathcal{F}_*$ of $\mathcal{M}$ such that $|\mathcal{F}_*| = 1$. They correspond to solved subproblems and they are not involved in step 2 any longer. Let us consider all the maximal consecutive groups of them. Let one such group be $\mathcal{F}_{i_g}, \mathcal{F}_{i_g+1}, \ldots, \mathcal{F}_{i_g+z}$, for some $\mathcal{F}_{i_g}$. We know that $\mathcal{F}_{i_g}$ is associated with just one rank $r_{i_{g'}}$, $g' \geq g$ and the only string in the corresponding subproblem is the one of rank $r_{i_{g'}}$ in $\mathcal{Z}$. Analogously, $\mathcal{F}_{i_g+1}$ is associated with just one rank $r_{i_{g'}+1}$ and so forth up to $\mathcal{F}_{i_g+z-1}$ included. Hence, for any such $\mathcal{F}_{i_g+w}$, we have that $|\mathcal{F}_{i_g+w}| + |\mathcal{M}_{i_g+w}| = \Delta_{i_{g'}+w}$. Therefore, because of Lemma 4, the term $-c\Delta_{i_{g'}+w} \log \Delta_{i_{g'}+w}$ of the wanted bound is accounted for. What we have said so far does not hold for $\mathcal{F}_{i_g+z}$ (unless it is the rightmost one of all the pivotal multisets $\mathcal{F}_*$). Let us deal with these cases later.

Let us now consider any pivotal multiset $\mathcal{F}_o$ created in the first execution of step 2 and containing more that one symbol. Let $\mathcal{P}_{z_o}$ the subproblem corresponding to $\mathcal{F}_o$. Let $r_p, r_{p+1}, \ldots, r_x$ be the ranks corresponding to $\mathcal{F}_o$ and let N_o be the number of symbols in $\mathcal{M}$ less than the ones in $\mathcal{F}_o$. Since $\mathcal{P}_{z_o}$ has offset size less than the one of $\mathcal{P}_0$, by inductive hypothesis we know that the algorithm retrieves the strings of $\mathcal{Z}$ with ranks $r_p, r_{p+1}, \ldots, r_x$ using time

$$c|\mathcal{F}_o|\log|\mathcal{F}_o| + c|\mathcal{F}_o| + c5L'_o - c\sum_{j=p}^{x-1}\Delta_j\log\Delta_j +$$

$$-c(r_p - N_o)\log(r_p - N_o) - c(|\mathcal{F}_o| - r_x + N_o + 1)\log(|\mathcal{F}_o| - r_x + N_o + 1)$$

where $L'_o + |\mathcal{P}_{z_o}|$ is the smallest number of symbols we need to probe to find the wanted strings amongst the ones in $\mathcal{P}_{z_o}$ (since *off*$_{z_o} = 2$, i.e. the first symbol of each string in $\mathcal{P}_{z_o}$ can be skipped). Let us call $-c(r_p - N_o)\log(r_p - N_o)$ the *left inductive term* and the two terms $c|\mathcal{F}_o|$, $c5L'_o$ *inductive remainder terms*. We will deal with them later.

By Lemma 4, we have term $-c\,(|\mathcal{F}_o| + |\mathcal{M}_o|)\log\,(|\mathcal{F}_o| + |\mathcal{M}_o|)$ from the first execution of step 2a. We can upper bound that term by the two-term expression $-c|\mathcal{F}_o|\log|\mathcal{F}_o| - c|\mathcal{M}_o|\log|\mathcal{M}_o|$. The first term $-c|\mathcal{F}_o|\log|\mathcal{F}_o|$ can be cancelled with the opposite term in the expression given by the inductive hypothesis.

For each rank $r_j \in \{r_p, r_{p+1}, \ldots, r_{x-1}\}$ the term $-c\Delta_j\log\Delta_j$ of the wanted upper bound is accounted for, since it is in the expression given by the inductive hypothesis. Amongst the ranks corresponding to $|\mathcal{F}_o|$, r_x is the one with which we have yet to deal.

Let us first assume that $|\mathcal{F}_{o+1}| = 1$. If that is true, we know that $N_* + |\mathcal{M}_o| = \Delta_x$, where we denoted with N_* the number of strings in $\mathcal{P}_{z_o}$ with rank (in $\mathcal{Z}$) greater than or equal to r_x. But $N_* = |\mathcal{F}_o| - r_x + N_o + 1$. Hence, to obtain the term $-c\Delta_x\log\Delta_x$ for the wanted bound, we need to mix the terms $-c(|\mathcal{F}_o| - r_x + N_o + 1)\log(|\mathcal{F}_o| - r_x + N_o + 1)$ (from the inductive hypothesis on $\mathcal{P}_{z_o}$) and $-c|\mathcal{M}_o|\log|\mathcal{M}_o|$ (from the first execution of step 2a).

It is easy to prove that for any $a, b \geq 1$, $(a+b)\log(a+b) \geq a\log a + b\log b + 2(a+b)$. Thus, we have that

$$-cN_*\log N_* - c|\mathcal{M}_o|\log|\mathcal{M}_o| \leq$$

$$-c(N_* + |\mathcal{M}_o|)\log(N_* + |\mathcal{M}_o|) + 2c(N_* + |\mathcal{M}_o|) = -c\Delta_x\log\Delta_x + 2c\Delta_x.$$

Thus, we have now obtained the $-c\Delta_x\log\Delta_x$ term for the last rank r_x of $\mathcal{F}_o$ for the case $|\mathcal{F}_{o+1}| = 1$. However, we still have to deal with the extra $2c\Delta_x$, let us call it the *extra remainder term*.

Let us now consider the case $|\mathcal{F}_{o+1}| > 1$. Thus we can use the inductive hypothesis and in this case $\mathcal{F}_{o+1}$ contributes to the bound a *left inductive term* $-c(r'_p - N'_o)\log(r'_p - N'_o)$ (analogous to the $-c(r_p - N_o)\log(r_p - N_o)$ for $\mathcal{F}_o$), where r'_p is the smallest rank associated with $\mathcal{F}_{o+1}$ and N'_o is the number of symbols in $\mathcal{M}$ less than the ones in $\mathcal{F}_{o+1}$. The term $-c\Delta_x\log\Delta_x$ is obtained in the same way we did before, only this time we have to mix three intervals instead of two, since in this case $\Delta_x = N_* + |\mathcal{M}_o| + (r'_p - N'_o)$. Hence, in this case the *extra remainder term* is $4c\Delta_x$.

Before we account for all the remainder terms, we still need to deal with the rightmost multiset of each maximal consecutive group of those multisets $\mathcal{F}_*$ such that $|\mathcal{F}_*| = 1$. The (single) ranks r_u of each one of these rightmost multisets are the only ones for which we have not yet obtained the term $-c\Delta_u\log\Delta_u$ of the wanted upper bound. For any such multiset $\mathcal{F}_i$ with rank r'_i ($i \leq i'$), $-c\Delta'_i\log\Delta'_i$ can be obtained in the exact same way we did above and hence we have an extra remainder term for each one of these ranks too.

Finally, let us account for the remainder terms. For each $\mathcal{F}_o$, we have at most three kinds of remainders: $c|\mathcal{F}_o|$, $c5L'_o$ and a third kind that is at most $4c\Delta_x$, where r_x is the largest rank associated with $\mathcal{F}_o$ and $L'_o + |\mathcal{P}_{z_o}|$ is the smallest number of symbols we need to probe to find the wanted strings amongst the ones in $\mathcal{P}_{z_o}$ (if any). Overall, the first and third kinds are upper bounded by $c5N$. By the definition of the second kind of remainder and since all the N symbols in $\mathcal{M}$ need to be probed to find the wanted strings, we have that $c5N + \sum_{\{\mathcal{F}_o \| \mathcal{F}_o| > 1\}} c5L'_o = c5L'$. ∎

7.2 Proof of Lemma 1

Let us consider the pruning phase. In the first step, the level-by-level visit of the skip tree of A can be easily done in $O(\|A\|)$: the skip tree contains only contact and branching subproblems, hence its has $O(\|A\|)$ nodes; also, by Lemma 5, all the calls to GROUP have a total cost which is linear in the number of columns of A, which is $\|A\|$. About the second step, the cost of SKIPVISIT on the skip tree is clearly $O(\|A\|)$. So is the cost of the third step where we do an $O(1)$ amount of work for each node in `Pruned`. In the fourth step we do $O(1)$ amount of work for each subproblem that is not in a subtree rooted at some $\mathcal{P}_{w_i} \in$ `Pruned`. By the definition of `Pruned` we already know that each one of the subproblems we touch in the fourth step will be later partitioned into two or more smaller subproblems. Thus they are certainly less than $\left|\bigcup_{i=1}^{d} A_i - A\right|$ (which is the total number of new subproblems that are created by the slicing of A). Thus the pruning phase takes $O\left(\|A\| + \left|\bigcup_{i=1}^{d} A_i - A\right|\right)$ time.

Let us consider the slicing phase. First of all, SLICEREC performs a depth first visit of the tree of A that whenever encounters a node in `Pruned` it does not go any deeper. Thus, the total number of nodes visited is equal to the number of subproblems that are partitioned into two or more subproblems by the slice operation.

Let us consider the internal nodes of A. For any such $\mathcal{P}_r$, in the first step, after the recursive calls to SLICEREC have returned, we call GROUP on a list containing $|\mathcal{C}_r| + \sum_{i=1}^{f} sub_{r_i}$ objects, where $\mathcal{C}_r$ is the set of contact suffixes of $\mathcal{P}_r$ (if it is a contact node) and sub_{r_i} is the number of subproblems into which the i-th child of $\mathcal{P}_r$ has been refined into (during the recursive call). We charge each sub_{r_i} term to the corresponding child. The second step costs $O(sub_r)$ time. The third step requires $O(1)$ amount of work for each one of the children of each new subproblem into which $\mathcal{P}_r$ has been partitioned: we charge any such $O(1)$ amount of work the the corresponding child. Thus, the total cost for an internal node $\mathcal{P}_r$ of A (including the costs charged to $\mathcal{P}_r$ by its parent) is $O(sub_r)$.

For each leaf $\mathcal{P}_r$ of A, the total work we do is of the order of the number of columns of $\mathcal{P}_r$ (by Lemma 5). The amount charged to $\mathcal{P}_r$ by its parent is of the same order. Thus, overall the cost for all the leaves of A, the internal nodes of A and the new nodes of each A_i is $O\left(\|A\| + \left|\bigcup_{i=1}^{d} A_i - A\right|\right)$.

Finally let us consider the finishing phase. In the first step we can use radix sorting and thus the first and second steps take $O\left(\left|\bigcup_{i=1}^{d} A_i - A\right|\right)$ time. The third step accesses each skip node of each new agglomerate $O(1)$ times, thus costing $O(\|A\|)$. The fourth step, accesses at most all the nodes that are not descendants of any node in `Pruned`. By using `Ranks`, each access to establish if a node is unsolved takes $O(1)$ time. Thus the total cost of the fourth step is $O\left(\left|\bigcup_{i=1}^{d} A_i - A\right|\right)$.

7.3 Proof of Theorem 3

At this point, Theorem 3 follows directly from the following Lemmas 16 and 17, whose proofs detail some of the ideas presented in Section 6.

Lemma 16 *The running time of the suffix multi-selection algorithm for a text of length N is upper bounded by $O\left(N\log N - \sum_{j=0}^{K} \Delta_j \log \Delta_j + N + rap\right)$, where rap is the total time required by all the* RAP*s minus the time for the* MSELMSET *calls.*

Proof: The cost of the initialization stage is dominated by the call to MSELMSET (building `Ranks` takes $O(N)$ time). By Lemma 4, we know that the cost of that call is within our target bound. After the Refine and Aggregate stage, the cost of the finalization stage (Section 5.6) is $O(N)$.

Let us now account for the contribution of the calls to MSELMSET to the total cost. Instead of the normal time cost, let us consider the *virtual symbol cost* of the calls to MSELMSET: a call to MselMset costs or (conceptually) *creates* x virtual symbols if the multiset that it receives in input has x objects. The virtual symbols created by the call to MselMset for some leading subproblem are exclusively created and "used" for that subproblem. Thus, unlike T's symbols, virtual symbols are not shared by subproblems. Naturally, virtual symbols form virtual strings: each subproblem $\mathcal{P}_i$ has associated $|\mathcal{P}_i|$ *virtual strings* that are made up of all the virtual symbols that will be created during the computation to refine $\mathcal{P}_i$ every time it is the leading subproblem of its current agglomerate. And, unlike the suffixes of T, all the virtual strings are *independent.*

Picking an unsolved agglomerate A and refining it with a rap can be seen as a two-part process: (a) picking an unsolved subproblem, the leading subproblem of A, and refining it with the call to MselMset; (b) refining all the other unsolved subproblems of A with the rest of the rap (mainly the call to Slice). Thus, if we take aside all the (b) parts of the raps and if we consider the subproblems to be subsets of virtual strings, then the suffix multi-selection algorithm behaves exactly like the multi-selection algorithm for independent strings in Section 7.1. Therefore, by Lemma 15, we have that the cost of the algorithm is $O\Big(N\log N - \sum_{j=0}^{K}\Delta_j\log\Delta_j + N + L + \mathit{rap}\Big)$, where $r_0 = 0$, $r_{K+1} = N+1$, $\Delta_j \equiv r_{j+1} - r_j$ for $0 \le j \le K$, L is the total number of virtual symbols (and *rap* is the total cost of the (b) parts of the rap)s.

Let us evaluate L. As we have seen above, a rap an agglomerate A creates a number of new virtual symbols that is equal to the cardinality of the multiset passed to MselMset (equal to $\|A\|$). Since it is not computed during MselMset call, the cardinality of such multiset must be $O(\mathit{rap}(A))$ (where $\mathit{rap}(A)$ denotes the total cost of the rap on A minus the cost of the MselMset call). Therefore $L = O(\mathit{rap})$. ■

Lemma 17 *$\mathit{rap} = O(N)$.*

Proof: Let us first evaluate the total cost of the steps of a rap on an agglomerate A minus the cost of the call to MselMset in the third step.

About the first step, to verify which kind of an agglomerate we are dealing with (generic or core cyclic) $O(1)$ scans of the contact node list of A (while using the `Suff` structure) are enough. Thus, the first step takes $O(\|A\|)$ time.

About the second step, the slightly more complex case is when A is core cyclic. In that case we first find find the columns $\langle c_j, r_j\rangle$ such that $T_{c_j-1} \in \mathcal{P}_i \notin A$, then their keys and from those we produce the keys for all the other columns. As we noticed, we do not actually access each suffix of A to assign it its key. We do that only with A's columns. Thus, the second step takes $O(\|A\|)$ time.

About the third step. Thanks to `Ranks`, retrieving the $|\mathcal{R}(\mathcal{P}_w)|$ ranks of the leading subproblem $\mathcal{P}_w$ of A takes $O(|\mathcal{R}(\mathcal{P}_w)|)$ time. As we said, there is a 1-to-1 correspondence between the leading suffixes and the root suffixes. Hence, retrieving the suffixes of $\mathcal{P}_w$'s and their keys takes $O(|\mathcal{P}_w|)$. We will deal with the cost of the call to MselMset later. Scanning the pivotal multisets and tagging the the columns after the call to MselMset clearly cost $O(\|A\|)$. If we exclude MselMset, the cost of the third step is $O(\|A\|)$ (since $|\mathcal{P}_w| = \|A\|$).

The cost of the fourth step is dominated by the cost of the call Slice(A) which, by Lemma 1, is $O(\|A\| + |\bigcup_{i=1}^{q} A_i - A|)$, where $A_1, \ldots, A_q$ are the agglomerates into which A is sliced.

Let us consider the fifth step and, in particular, its three substeps described in Section 5.2. In the first substep, we call LeadVisit on the root of each agglomerate in `undecided_g`. LeadVisit accesses the nodes of the skip tree of the agglomerate it received in input and for each node does $O(1)$ time worth of work. In the second substep we compute the prefix sum array $\texttt{SCon}_i$ of $\texttt{Con}_i$ for each $A_i \in$ `undecided_g` and then we tag each column of A_i. So both substeps require $O\Big(\sum_{\{A_i \in \texttt{undecided\_g}\}} \|A_i\|\Big)$ time. In the third substep we call Slice(A_i) for each $A_i \in$ `undecided_g`. All those calls clearly dominates the total cost of the substep. For each $A_i \in$ `undecided_g`, let

$A_{i_1}, \ldots, A_{i_{t(i)}}, A_{i_{t(i)+1}}$ be the $t(i)+1$ agglomerates into which A_i is sliced (for simplicity's sake let us assume that there is a $A_{i_{t(i)+1}}$, i.e. an exhausted agglomerate, for each A_i). By Lemma 1, the total cost of the third substep is $O\Big(\sum_{\{A_i \in \texttt{undecided\_g}\}} \Big(\|A_i\| + \Big|\bigcup_{j=1}^{t(i)+1} A_{i_j} - A_i\Big|\Big)\Big)$.

Finally, in the sixth step we operate on each $A_i \in \texttt{joinable\_g}$. We have two cases in which we either do a Join or a SliceJoin between A_i and the agglomerate A_{i*} with which A_i is joinable. In the second case, let A_{i*1} be the agglomerate "sliceable" from A_{i*} such that $\langle c_j, r_j \rangle$ is a column of A_{i*1} iff $T_{c_j - 1} \in \mathcal{P}_z \in A_i$. By Lemmas 2 and 3 the total cost of the sixth step is $O\Big(\sum_{\{A_i \in \texttt{joinable\_g}\}} \|A_i\|\Big)$ in the first case and $O\Big(\sum_{\{A_i \in \texttt{joinable\_g}\}} \|A_i\| + |A_{i*1}|\Big)$ in the second. Let us denote with `slicejoinable_g` all the A_i's in `joinable_g` that need a SliceJoin in the eighth step.

Excluding the cost of the call to MselMset in the third step and, since $\sum_{\{A_i \in \texttt{undecided\_g}\}} \|A_i\| + \sum_{\{A_i \in \texttt{joinable\_g}\}} \|A_i\| \leq \|A\|$ (in the fourth step some of the A_i's sliced from A may have been moved to `unsolved_g`), the total time for the rap is

$$O(\|A\| + \Gamma + \Lambda + \Phi)$$

$$\Gamma = \left|\bigcup_{i=1}^{q} A_i - A\right| \qquad \Lambda = \sum_{\{A_i \in \texttt{undecided\_g}\}} \left|\bigcup_{j=1}^{t(i)+1} A_{i_j} - A_i\right| \qquad \Phi = \sum_{\{A_i \in \texttt{slicejoinable\_g}\}} |A_{i*1}|$$

To complete the analysis of *rap* let us introduce the *events*. During the algorithm five kinds of crucial events happen. (a) *Column fusions*: when during a Join(A', A) (or at the end of a SliceJoin) a column of $\langle c_i, r_i \rangle$ of A' is fused with the column $\langle c_j, r_j \rangle$ of A such that $r_i + 1 = c_j$, to form the column $\langle c_i, r_j \rangle$. (b) *Subproblem creations*: when during a Slice (or at the beginning of a SliceJoin) some subproblem $\mathcal{P}_j$ is partitioned into smaller subproblems $\mathcal{P}_{i_1}, \ldots, \mathcal{P}_{i_p}$, we have p *subproblem creation events.* (c) *Suffix exhaustions*: when during the third step of a rap for A, after the MselMset call, some suffix $T_e \in \mathcal{P}_w \in A$, where $\mathcal{P}_w$ is the leading subproblem of A, becomes exhausted (i.e. T_e belongs to a pivotal multiset $\mathcal{M}_i$, thus we know for sure that it is not one of the wanted suffixes). (d) *Suffix discoveries*: when during the third step of a rap for A, some suffix T_w in $\mathcal{P}_w$, the leading subproblem of A, is recognized as one of the wanted suffixes (i.e. T_w belongs to a pivotal multiset $\mathcal{F}_i$ such that $|\mathcal{F}_i| = 1$). (e) *Inner collisions*: when during the second step of a rap for A, we encounter a column $\langle c_i, r_i \rangle$ of A such that $\langle c_j, r_j \rangle$, where $c_j = r_i + 1$, is also in A while T_{c_i} and T_{c_j} do not belong to the same subproblem.

A column is never divided into smaller ones and a subproblem is never merged with others to form a larger subproblem. Also, when a suffix becomes exhausted or a wanted suffix is discovered, they can never be in an unsolved subproblem again. Finally, the same inner collision cannot be repeated after the rap in which it has been detected has ended, since the two columns colliding will not be part of the same agglomerate any longer. By all the above, it is easy to see that one particular event cannot be repeated twice. Since there are N suffixes and we start with N columns and one subproblem, we can conclude that the total number of events during the computation is $O(N)$, or $\leq 6N$ to be precise.

Let us establish how many events take place *during* a generic rap for an agglomerate A. Let us start with *subproblem creations.* In the fourth step, for each agglomerate A_i sliced from A, we have $|A_i - A|$ subproblem creations, one for each new (i.e. not coming from A) subproblem in A_i. Analogously, in the third substep of the fifth step (Section 5.2), for each $A_i \in \texttt{undecided\_g}$ and for each A_{i_j} sliced from A_i, we have $\left|A_{i_j} - A_i\right|$ subproblem creations. Finally, in the eighth step, for each $A_i \in \texttt{slicejoinable\_g}$, we have $|A_{i*1}|$ subproblem creations. Summing up, $\Gamma + \Lambda + \Phi$ subproblem creations take place during the rap for an agglomerate A. The total number of *suffix discoveries* and *suffix exhaustions* occurring during the rap on A is equal to the total number of columns of the agglomerates that in the fourth step are moved either to `exhausted_g` or to `undecided_g`. Each column of each agglomerate moved to `undecided_g` corresponds to a *suffix exhaustion.* Each column of each agglomerate moved to `exhausted_g` corresponds to either a *suffix discovery* or a *suffix exhaustion.* Analogously, the numbers of *column fusions* and *inner collisions* during the RAP of A are equal to the total numbers of columns of the A_i's that in the fourth step are moved to `joinable_g` and `unsolved_g`, respectively. Since $\sum_{i=1}^{q}\|A_i\| = \|A\|$, we can conclude that the *total number of events* occurring during the RAP for A is $\|A\|+\Gamma+\Lambda+\Phi$. As we have seen, if we exclude the calls to MSELMSET, the cost of the RAP for A is $O(\|A\|+\Gamma+\Lambda+\Phi)$. Therefore the total time required by all the RAPs minus the time for the MSELMSET calls is of the order of the total number of events, which is $O(N)$. ■

7.4 Proof of Theorem 1

Theorem 1 follows directly from Lemmas 18 and 19.

Lemma 18 *Given a text T of N symbols drawn from an unbounded alphabet and K consecutive ranks $r_1, \ldots, r_K$ (i.e. $r_2 = r_1 + 1, r_3 = r_2 + 1, \ldots, r_K = r_{K-1} + 1$), the K text suffixes of ranks $r_1, \ldots, r_K$ can be found using $O(K \log K + N)$ time and comparisons.*

Proof: To retrieve the wanted suffixes in $O(K \log K + N)$ time we first apply the suffix multi-selection algorithm on T with only r_1 and r_K. Then we go through an *intermediate stage* that takes the subproblems left by the suffix multi-selection and prepares them for a second suffix multi-selection. After that we apply again the suffix multi-selection on T but this time (a) we use all the ranks $r_1, \ldots, r_K$ and (b) we skip the *initialization stage* (because of the work done in the intermediate stage).

Let us give the details of the process.

We execute the suffix multi-selection algorithm on T with $\mathcal{R} = \{r_1, r_K\}$. After the computation ends, we have the two wanted suffixes, let them be T_l and T_r ($T_l < T_r$), the exhausted agglomerates and a subproblem list `SubList` $= \langle \mathcal{P}_0, \ldots, \mathcal{P}_{i_l}, \ldots, \mathcal{P}_{i_r}, \ldots, \mathcal{P}_p \rangle$, ($i_l < i_r < p$), with the following properties. (a) $\mathcal{P}_{i_l} = \{T_l\}$ and $\mathcal{P}_{i_r} = \{T_r\}$ (they are the only solved subproblems). (b) $\mathcal{P}_i < \mathcal{P}_{i_l}$, $\mathcal{P}_{i_l} < \mathcal{P}_{i'} < \mathcal{P}_{i_r}$ and $\mathcal{P}_{i_r} < \mathcal{P}_{i''}$, for each $i < i_l$, $i_l < i' < i_r$ and $i'' > i_r$, respectively. (c) All together the subproblems $\mathcal{P}_{i'}$ with $i_l < i' < i_r$ contains *exactly* $K-2$ suffixes and they are the ones with ranks $r_2, \ldots, r_{K-1}$ (but for each of those suffixes we do not know which $r_2, \ldots, r_{K-1}$ is its rank).

The *intermediate stage* has the following steps.

First, for each subproblem in `SubList` we retrieve its suffixes. Recall that for each agglomerate A_i only contact and root suffixes are explicitly stored during the computation. To retrieve all the suffixes of each $\mathcal{P}_j \in A_i$ we simply need to visit the tree of A_i from its root with SUFFIXVISIT($\mathcal{P}_r$) defined as follows. (i) If $\mathcal{P}_r$ is a leaf then $\mathcal{P}_r = \{T_c \mid \langle c, r\rangle$ is in $\mathcal{P}_r$'s column list$\}$. After we retrieved the suffixes of $\mathcal{P}_r$ we return the set $\{T_{i+1} \mid T_i \in \mathcal{P}_r\}$. ($ii$) Otherwise, if $\mathcal{P}_r$ is a contact node (but not a leaf) we retrieve $Con = \{T_c \mid \langle c, r\rangle$ is in $\mathcal{P}_r$'s column list$\}$. (iii) In any case, we call SUFFIXVISIT($\mathcal{P}_{r_i}$) for each children $\mathcal{P}_{r_i}$ of $\mathcal{P}_r$, let $\mathcal{S}$ be the set of all the suffixes we receive from all these recursive calls. (iv) Then, we set $\mathcal{P}_r = Con \cup \mathcal{S}$ and we return the set $\{T_{i+1} \mid T_i \in \mathcal{P}_r\}$.

Second, with a scan of `SubList`, we do the following. We merge $\mathcal{P}_0$ with all the subproblems in its neighborhood $\mathcal{N}_0$, let them be $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_{n_0}$, into one (the subproblems in the same neighborhood are adjacent in `SubList`). We do the same for $\mathcal{P}_{n_0+1}$ and its neighborhood $\mathcal{P}_{n_0+2}, \ldots, \mathcal{P}_{n_1}$, then for $\mathcal{P}_{n_1+1}$ and so forth until all the subproblems $\mathcal{P}_i < \mathcal{P}_{i_l}$ have been treated. After that we do the same for all the subproblems $\mathcal{P}_{''} > \mathcal{P}_{i_r}$. Finally we merge all the subproblems $\mathcal{P}_{i'}$ with $i_l < i' < i_r$ into one, let it be $\mathcal{P}_{\#}$. After the first step, the new `SubList` $= \left\langle \mathcal{P}_0, \ldots, \mathcal{P}_{i_l}, \mathcal{P}_{\#}, \mathcal{P}_{i_r}, \ldots, \mathcal{P}_{p'} \right\rangle$ maintains the same properties of the original one. However, the meaning of *the integer labels* of the subproblems may have changed. Now for each subproblem $\mathcal{P}_i \in$ `SubList`, ℓ_i is the number of suffixes of T that are lexicographically smaller than each $T_j \in \mathcal{P}_i$ (because now for each $\mathcal{P}_i$ we have that $\mathcal{N}_i = \mathcal{P}_i$, whereas originally that was guaranteed only for $\mathcal{P}_{i_l}$ and $\mathcal{P}_{i_r}$).

Third, for each $T_i \in \mathcal{P}_{\#}$, let T_i's *key* be its first symbol $T[i]$. We call $\textsc{MselMset}(\mathcal{P}_{\#}, r_2, \ldots, r_{K-1})$ and we obtain $\mathcal{P}_{\#}$'s pivotal subsets $\mathcal{M}_0, \mathcal{F}_1, \mathcal{M}_1, \ldots, \mathcal{F}_t, \mathcal{M}_t$. Since we know that $\mathcal{P}_{\#}$ contains all and only the $K-2$ suffixes with ranks $r_2, \ldots, r_{K-1}$, all the $\mathcal{M}_i$ *are void.* Thus, from each $\mathcal{F}_i$ we create a subproblem $\mathcal{P}_{\#i}$ with the following properties: (a) $|\mathcal{P}_{\#i}| = |\mathcal{R}(\mathcal{P}_{\#i})|$ and (b) $\ell_{\#i}$ is the number of suffixes of T smaller than each one in $\mathcal{P}_{\#i}$. After this step we have SubList $= \langle \mathcal{P}_0, \ldots, \mathcal{P}_{i_l}, \mathcal{P}_{\#1}, \ldots, \mathcal{P}_{\#t}, \mathcal{P}_{i_r}, \ldots, \mathcal{P}_{p'} \rangle$.

Fourth, from each $\mathcal{P}_i$ with $i < i_l$ or $i > i_r$ (they are all exhausted) we make an agglomerate A_i and we move it to exhausted_g. We do the same for $\mathcal{P}_{i_l}$ and $\mathcal{P}_{i_r}$ (although, as subproblems, they are solved ones). Finally, from each $\mathcal{P}_{\#j}$ we make an agglomerate $A_{\#i}$ and we move it to unsolved_g.

After the intermediate stage, we apply again the suffix multi-selection algorithm on T with the full rank set $\mathcal{R} = \{r_1, \ldots, r_K\}$ in the following way. We skip the initialization stage completely and we start immediately with the refine and aggregate stage (using SubList, unsolved_g, exhausted_g, the agglomerates and the subproblems we already have after the intermediate stage). The rest of the computation proceeds normally.

Let us now evaluate the cost of the whole computation. The first call of the suffix multi-selection algorithm is done with just two ranks, r_1 and r_K. Thus, by Theorem 3, its cost is $O(N)$. Let us consider the intermediate stage. The first step requires $O(N)$ time: for each agglomerate A_i, SuffixVisit costs $O(suf_i)$ where $suf_i = \sum_{\{\mathcal{P}_j \in A_i\}} |\mathcal{P}_j|$. For the second step a scan of SubList is enough and the cost is $O(N)$. The cost of the third step si dominated by the cost of the call to MselMset. Since it is done on a multiset of $K-2$ elements (and with a set of $K-2$ ranks), its cost is clearly $O(K \log K)$. Finally, an $O(N)$ time scan of SubList is enough for the fourth step.

Let us now consider the cost of the second execution of the suffix multi-selection. The complexity proof is the same of the one for Theorem 3 except for two aspects. First, there is no initialization stage, thus the cost of finding the pivotal multisets of$\{T[1], \ldots, T[N]\}$ disappears. Second, the total contribution of the MselMset calls made during the whole refine and aggregate stage changes as follows. The total number of suffixes of all the leading subproblems is at most $K-2$. Thus the total number of virtual strings is at most $K-2$. On the other hand, the events that take place during all the raps are still $O(N)$. Thus, the total number of virtual symbols in the virtual strings is $O(N)$. All the other additional costs of the raps remain the same. Therefore, by Lemma 15, we have that the total cost of the second call of the suffix multi-selection is is $O(K \log K + N)$. ■

For any two subproblems $\mathcal{P}_i$ and $\mathcal{P}_{i'}$ such that $\mathcal{N}_i \neq \mathcal{N}_{i'}$, let $lcp(\mathcal{P}_i, \mathcal{P}_{i'})$ be the length of the longest common prefix of any two suffixes $T_{j_i} \in \mathcal{P}_i$ and $T_{j_{i'}} \in \mathcal{P}_{i'}$. Let us extend the notation to neighborhoods: for any two $\mathcal{N}_i \neq \mathcal{N}_{i'}$, $lcp(\mathcal{N}_i, \mathcal{N}_{i'})$ is equal to $lcp(\mathcal{P}_i, \mathcal{P}_{i'})$. We have the following:

Lemma 19 *The suffix multi-selection algorithm can be modified (without changing its asymptotical time complexity) so that $lcp(\mathcal{P}_i, \mathcal{P}_{i'})$ can be computed in $O(1)$ time, for any two $\mathcal{P}_i, \mathcal{P}_{i'}$ such that $\mathcal{N}_i \neq \mathcal{N}_{i'}$.*

Proof: As we have seen, for any subproblem $\mathcal{P}_i$, the subproblems in its neighborhood $\mathcal{N}_i$ are in contiguous positions in SubList. We maintain another list NeighList whose elements represent the neighborhoods: the i-th element in NeighList represents the i-th contiguous group of subproblems in SubList forming a neighborhood. Each $\mathcal{P}_i$ in SubList has a link to the element in NeighList corresponding to its neighborhood. We also maintain a third list LcpList such that if $\mathcal{P}_i$ and $\mathcal{P}_{i'}$ are in two adjacent neighborhoods NeighList$[j]$ and NeighList$[j+1]$ then LcpList$[j] = lcp(\mathcal{P}_i, \mathcal{P}_{i'})$. A cartesian tree is maintained on LcpList and a structure for dynamic LCA queries (e.g. [5]) is maintained on the cartesian tree.

SubList, NeighList and LcpList are updated in the finishing phase of the SLICE operation (see Section 5.3.3). As we have seen, a subproblem $\mathcal{P}_i$ in SubList is replaced by a partitioning $\mathcal{P}_{i_1}, \ldots, \mathcal{P}_{i_r}$ of it. If $\mathcal{P}_i$ was active then the partitioning is necessarily a refining one, otherwise we do not care. Thus, if $\mathcal{P}_i$ was inactive, the set of the suffixes whose subproblem belong to $\mathcal{N}_i$ does not change and neither NeighList nor LcpList needs to be updated.

On the other hand, if $\mathcal{P}_i$ was active then $\mathcal{N}_i = \mathcal{P}_i$ and $\mathcal{N}_{i_j} = \mathcal{P}_{i_j}$, for each $1 \leq j \leq r$. Thus, the element in NeighList for $\mathcal{N}_i$ is replaced by the ones for $\mathcal{N}_{i_1}, \ldots, \mathcal{N}_{i_r}$. Since the partitioning is a refining one, we have that $lcp(\mathcal{N}_p, \mathcal{N}_{i_1}) = lcp(\mathcal{N}_p, \mathcal{N}_i)$ and $lcp(\mathcal{N}_{i_r}, \mathcal{N}_s) = lcp(\mathcal{N}_i, \mathcal{N}_s)$, where $\mathcal{N}_p$ and $\mathcal{N}_s$ are (were) the predecessor and successor of $\mathcal{P}_i$ in NeighList. Thus the corresponding entries in LcpList do not need to be updated (and neither does the cartesian tree nor the structure for LCA queries). The values $lcp(\mathcal{N}_{i_1}, \mathcal{N}_{i_2}), \ldots, lcp\big(\mathcal{N}_{i_{r-1}}, \mathcal{N}_{i_r}\big)$ can be easily found by doing $r-1$ LCA queries. Since $\mathcal{P}_i$ has been refined, we know that none of the values $lcp(\mathcal{N}_{i_1}, \mathcal{N}_{i_2}), \ldots, lcp\big(\mathcal{N}_{i_{r-1}}, \mathcal{N}_{i_r}\big)$ can be smaller than either $lcp(\mathcal{N}_p, \mathcal{N}_i)$ or $lcp(\mathcal{N}_i, \mathcal{N}_s)$. Thus each pair of insertions of $\mathcal{N}_{i_j}$ in NeighList and of $lcp\big(\mathcal{N}_{i_j}, \mathcal{N}_{i_{j+1}}\big)$ in LcpList corresponds to the insertion of a leaf on the cartesian tree built on LcpList. This kind of updates of the structure for LCA queries can be done in $O(1)$ time (hence adding an extra $O(N)$ term to the complexity bound for the suffix multi-selection algorithm). ■

8 Conclusions

We studied partial compression and text indexing problems, and as a technical piece, the suffix multi-selection problem. The main theme is that when comparing an arbitrary set of suffixes which might overlap in sophisticated ways, we need to devise methods to avoid rescanning characters to get optimal results. We achieve this with a variety of structural observations and carefully arranging computations, and achieve bounds optimal with respect to those known for atomic elements. Other partial suffix problems will be of great interest in Stringology and its applications.

References

[1] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A comparison of sorting algorithms for the Connection Machine CM-2. *C.ACM*, 39:273–297, 1996.

[2] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, August 1973.

[3] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Research Report 124, Digital SRC, Palo Alto, CA, USA, May 1994.

[4] J. M. Chambers. Partial sorting (algorithm 410). *Commun. ACM*, 14(5):357–358, 1971.

[5] Richard Cole and Ramesh Hariharan. Dynamic LCA queries on trees. *SIAM Journal on Computing*, 34(4):894–923, August 2005.

[6] W. Cunto and J. I. Munro. Average case selection. *Journal of the ACM*, 36(2):270–279, April 1989.

[7] D. Dobkin and I. Munro. Optimal time minimal space selection algorithms. *Journal of the ACM*, 28(3):454–461, July 1981.

[8] M. Farach. Optimal suffix tree construction with large alphabets. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 137–143. IEEE Computer Society Press, 1997.

[9] Martin Farach-Colton, Paolo Ferragina and S. Muthukrishnan. On the sorting-complexity of suffix tree construction *J. ACM*, 47(6):987–1011, 2000.

[10] R. W. Floyd and R. L. Rivest. Expected time bounds for selection. *C.ACM*, 18(3):165–172, 1975.

[11] G. Franceschini, R. Grossi, and S. Muthukrishnan. Optimal cache-aware suffix selection. In *STACS*, volume 3, pages 457–468, 2009.

[12] G. Franceschini and S. Muthukrishnan. Optimal suffix selection. In *STOC*, pages 328–337. ACM, 2007.

[13] C. A. R. Hoare. Algorithm 65: Find. *Communications of the ACM*, 4(7):321–322, July 1961.

[14] H.-K. Hwang and T.-H. Tsai. Quickselect and the Dickman function. *Comb.,Prob.&Comp.*, 11(4), 2002.

[15] K. Kaligosi, K. Mehlhorn, J. I. Munro, and P. Sanders. Towards optimal multiple selection. *ICALP* 2005, *LNCS* volume 3580, 103–114, 2005.

[16] J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *J.ACM*, 53(6):918–936, 2006.

[17] J. Kärkkäinen and E. Ukkonen. Sparse suffix trees. *LNCS*, 1090:219, 1996.

[18] D.E. Knuth. The Art of Computer Programming, vol. 3. *Addison-Wesley*, 1998.

[19] M. Kuba. On quickselect, partial sorting and multiple quickselect. *IPL*, 99(5):181–186, 2006.

[20] H. M. Mahmoud, R. Modarres, and R. T. Smythe. Analysis of quickselect: An algorithm for order statistics. *Informatique théorique et applications*, 29(4), 1995.

[21] Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, October 1993.

[22] Giovanni Manzini. An analysis of the Burrows-Wheeler transform. *J. ACM*, 48(3):407–430, 2001.

[23] C. Martinez, D. Panario, and A. Viola. Adaptive sampling for quickselect. *SODA* 2004, 447–455, 2004.

[24] E. M. McCreight. A space-economical suffix tree construction algorithm. *J.ACM*, 23(2):262–272, 1976.

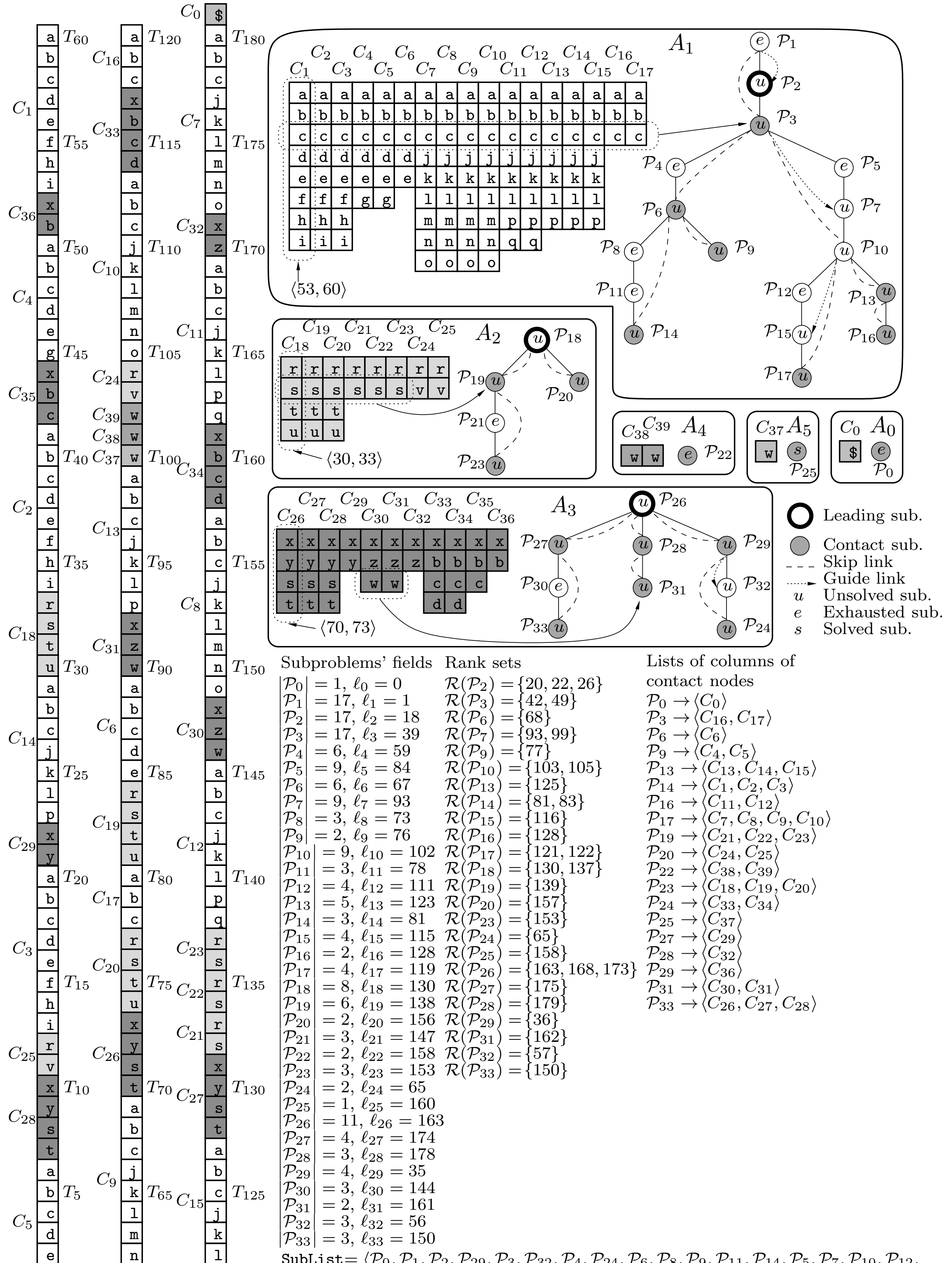
[25] J. Ian Munro and Venkatesh Raman. Sorting multisets and vectors in-place. *WADS* 1991: 473-480.

[26] A. Panholzer. Analysis of multiple quickselect variants. *TCS*, 302(1–3):45–91, 2003.

[27] I. Pohl. A sorting problem and its complexity. *C.ACM*, 15(6):462–464, June 1972.

[28] H. Prodinger. Multiple Quickselect—Hoare's Find algorithm for several elements. *IPL*, 56:123, 1995.

[29] P. Weiner. Linear pattern matching algorithms. In *FOCS'73*, 1–11, 1973.



Subproblems' fields

$|\mathcal{P}_0| = 1$, $\ell_0 = 0$
$|\mathcal{P}_1| = 17$, $\ell_1 = 1$
$|\mathcal{P}_2| = 17$, $\ell_2 = 18$
$|\mathcal{P}_3| = 17$, $\ell_3 = 39$
$|\mathcal{P}_4| = 6$, $\ell_4 = 59$
$|\mathcal{P}_5| = 9$, $\ell_5 = 84$
$|\mathcal{P}_6| = 6$, $\ell_6 = 67$
$|\mathcal{P}_7| = 9$, $\ell_7 = 93$
$|\mathcal{P}_8| = 3$, $\ell_8 = 73$
$|\mathcal{P}_9| = 2$, $\ell_9 = 76$
$|\mathcal{P}_{10}| = 9$, $\ell_{10} = 102$
$|\mathcal{P}_{11}| = 3$, $\ell_{11} = 78$
$|\mathcal{P}_{12}| = 4$, $\ell_{12} = 111$
$|\mathcal{P}_{13}| = 5$, $\ell_{13} = 123$
$|\mathcal{P}_{14}| = 3$, $\ell_{14} = 81$
$|\mathcal{P}_{15}| = 4$, $\ell_{15} = 115$
$|\mathcal{P}_{16}| = 2$, $\ell_{16} = 128$
$|\mathcal{P}_{17}| = 4$, $\ell_{17} = 119$
$|\mathcal{P}_{18}| = 8$, $\ell_{18} = 130$
$|\mathcal{P}_{19}| = 6$, $\ell_{19} = 138$
$|\mathcal{P}_{20}| = 2$, $\ell_{20} = 156$
$|\mathcal{P}_{21}| = 3$, $\ell_{21} = 147$
$|\mathcal{P}_{22}| = 2$, $\ell_{22} = 158$
$|\mathcal{P}_{23}| = 3$, $\ell_{23} = 153$
$|\mathcal{P}_{24}| = 2$, $\ell_{24} = 65$
$|\mathcal{P}_{25}| = 1$, $\ell_{25} = 160$
$|\mathcal{P}_{26}| = 11$, $\ell_{26} = 163$
$|\mathcal{P}_{27}| = 4$, $\ell_{27} = 174$
$|\mathcal{P}_{28}| = 3$, $\ell_{28} = 178$
$|\mathcal{P}_{29}| = 4$, $\ell_{29} = 35$
$|\mathcal{P}_{30}| = 3$, $\ell_{30} = 144$
$|\mathcal{P}_{31}| = 2$, $\ell_{31} = 161$
$|\mathcal{P}_{32}| = 3$, $\ell_{32} = 56$
$|\mathcal{P}_{33}| = 3$, $\ell_{33} = 150$

Rank sets

$\mathcal{R}(\mathcal{P}_2) = \{20, 22, 26\}$
$\mathcal{R}(\mathcal{P}_3) = \{42, 49\}$
$\mathcal{R}(\mathcal{P}_6) = \{68\}$
$\mathcal{R}(\mathcal{P}_7) = \{93, 99\}$
$\mathcal{R}(\mathcal{P}_9) = \{77\}$
$\mathcal{R}(\mathcal{P}_{10}) = \{103, 105\}$
$\mathcal{R}(\mathcal{P}_{13}) = \{125\}$
$\mathcal{R}(\mathcal{P}_{14}) = \{81, 83\}$
$\mathcal{R}(\mathcal{P}_{15}) = \{116\}$
$\mathcal{R}(\mathcal{P}_{16}) = \{128\}$
$\mathcal{R}(\mathcal{P}_{17}) = \{121, 122\}$
$\mathcal{R}(\mathcal{P}_{18}) = \{130, 137\}$
$\mathcal{R}(\mathcal{P}_{19}) = \{139\}$
$\mathcal{R}(\mathcal{P}_{20}) = \{157\}$
$\mathcal{R}(\mathcal{P}_{23}) = \{153\}$
$\mathcal{R}(\mathcal{P}_{24}) = \{65\}$
$\mathcal{R}(\mathcal{P}_{25}) = \{158\}$
$\mathcal{R}(\mathcal{P}_{26}) = \{163, 168, 173\}$
$\mathcal{R}(\mathcal{P}_{27}) = \{175\}$
$\mathcal{R}(\mathcal{P}_{28}) = \{179\}$
$\mathcal{R}(\mathcal{P}_{29}) = \{36\}$
$\mathcal{R}(\mathcal{P}_{31}) = \{162\}$
$\mathcal{R}(\mathcal{P}_{32}) = \{57\}$
$\mathcal{R}(\mathcal{P}_{33}) = \{150\}$

Lists of columns of contact nodes

$\mathcal{P}_0 \rightarrow \langle C_0\rangle$
$\mathcal{P}_3 \rightarrow \langle C_{16}, C_{17}\rangle$
$\mathcal{P}_6 \rightarrow \langle C_6\rangle$
$\mathcal{P}_9 \rightarrow \langle C_4, C_5\rangle$
$\mathcal{P}_{13} \rightarrow \langle C_{13}, C_{14}, C_{15}\rangle$
$\mathcal{P}_{14} \rightarrow \langle C_1, C_2, C_3\rangle$
$\mathcal{P}_{16} \rightarrow \langle C_{11}, C_{12}\rangle$
$\mathcal{P}_{17} \rightarrow \langle C_7, C_8, C_9, C_{10}\rangle$
$\mathcal{P}_{19} \rightarrow \langle C_{21}, C_{22}, C_{23}\rangle$
$\mathcal{P}_{20} \rightarrow \langle C_{24}, C_{25}\rangle$
$\mathcal{P}_{22} \rightarrow \langle C_{38}, C_{39}\rangle$
$\mathcal{P}_{23} \rightarrow \langle C_{18}, C_{19}, C_{20}\rangle$
$\mathcal{P}_{24} \rightarrow \langle C_{33}, C_{34}\rangle$
$\mathcal{P}_{25} \rightarrow \langle C_{37}\rangle$
$\mathcal{P}_{27} \rightarrow \langle C_{29}\rangle$
$\mathcal{P}_{28} \rightarrow \langle C_{32}\rangle$
$\mathcal{P}_{29} \rightarrow \langle C_{36}\rangle$
$\mathcal{P}_{31} \rightarrow \langle C_{30}, C_{31}\rangle$
$\mathcal{P}_{33} \rightarrow \langle C_{26}, C_{27}, C_{28}\rangle$

SubList= $\langle \mathcal{P}_0, \mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_{29}, \mathcal{P}_3, \mathcal{P}_{32}, \mathcal{P}_4, \mathcal{P}_{24}, \mathcal{P}_6, \mathcal{P}_8, \mathcal{P}_9, \mathcal{P}_{11}, \mathcal{P}_{14}, \mathcal{P}_5, \mathcal{P}_7, \mathcal{P}_{10}, \mathcal{P}_{12}, \mathcal{P}_{15}, \mathcal{P}_{17}, \mathcal{P}_{13}, \mathcal{P}_{16}, \mathcal{P}_{18}, \mathcal{P}_{19}, \mathcal{P}_{30}, \mathcal{P}_{21}, \mathcal{P}_{33}, \mathcal{P}_{23}, \mathcal{P}_{20}, \mathcal{P}_{22}, \mathcal{P}_{25}, \mathcal{P}_{31}, \mathcal{P}_{26}, \mathcal{P}_{27}, \mathcal{P}_{28}\rangle$

Figure 2: "Snapshot" of the computation for a text T with $N = 181$ symbols and $K = 34$ ranks in $\mathcal{R}$. Agglomerates A_1, A_2, and A_3 are unsolved, while A_0, A_4 and A_5 are exhausted. The columns of each agglomerate are pictured beside it (as contiguous substrings of T). T is pictured as a partitioning of the columns of the agglomerates.